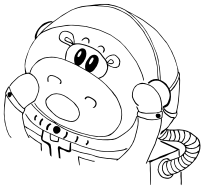
Korespondenční Seminář z Programování

33. ročník **KSP** **Září 2020**

Milí řešitelé, řešitelky a řešitelčata!

Dostává se k vám první číslo hlavní kategorie 33. ročníku KSP.

Letos se můžete těšit v každé z pěti sérií hlavní kategorie na: **4 normální úlohy**, z toho alespoň jedna praktická opendatová. Také na **grafický seriál** a kuchařky na nějaké zajímavé informatické témata, hodící se k úlohám dané série. Občas se nám také objeví **bonusová X-ková úloha**, za kterou lze získat X-kové body.



Oproti loňskému ročníku se do výsledkové listiny započítávají všechny úlohy a body se již nepřepočítávají podle počtu vyřešených sérií.

Autorská řešení úloh budeme vystavovat hned po skončení série. Pokud nás pak při opravování napadnou nějaké komentáře k řešením od vás, zveřejníme je dodatečně.

Odměny & na Matfyz bez přijímaček

Za úspěšné řešení KSP můžete být **přijati na MFF UK bez přijímacích zkoušek**. Úspěšným řešitelem se stává ten, kdo získá za celý ročník (této kategorie) alespoň 50 % bodů. Za letošní rok půjde získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150. Maturanti pozor, pokud chcete prominutí využít letos, musíte to stihnout do konce čtvrté série, pátá už bude moc pozdě. Také každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok, placku a možná i další překvapení.

Termín série: **1. listopadu 2020 ve 32:00**

Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/submit/*

Značky úloh: Lehčí úloha (či její část) vhodná pro začátečníky; Praktická open-data úloha; Úloha, u které doporučujeme začíst se do kuchařky; Seriálová úloha

Odměna série: Každému, kdo vyřeší 4 úlohy alespoň na polovinu bodů, pošleme **sladkou odměnu**.

První série třicátého třetího ročníku KSP

Letos nebudou mít úlohy žádný velký spojující příběh, na který jste mohli být zvyklí z minulých let. Namísto toho budou mít jednotlivé úlohy své vlastní malé příběhy. Snad se vám budou líbit :)

33-1-1 Prosperující provincie — 10 bodů

Kevin si z dlouhé chvíle listuje historickou knihou. Vypozoroval, že existovaly říše, které ovládaly provincie. Kontrola provincií se měnila historickými událostmi.

Historické události lze rozdělit do čtyř typů:

- **Osídlení provincie** znamená, že provincii začne ovládat kolonizující říše (podobně jako Korinth osídlil Syrakusy) a pro nás to vlastně znamená vznik civilizované kolonie. Každá provincie musí být osídlena právě jednou.
- **Katastrofa v provincii** znamená její zánik. Provincie je po katastrofě do konce dějin neobyvatelnou (například jako výbuch v Pompejích).
- **Podrobení říše** znamená, že všechny provincie ovládané podrobenou říší začne ovládat říše, která si ji podrobila. Podrobená říše de facto přestane existovat a neúčastní se dalších událostí (podobně jako Makedonie ovládla Egypt).
- **Zničení říše** znamená, že všechny provincie ovládané zničenou říší budou vypáleny a zasypány solí (podobně jako Řím zničil Kartágo). Zničené provincie už není možné znovu kolonizovat a zničená říše už se neúčastní dalších událostí.

Je zaručeno, že každá provincie zanikne (ať už katastrofou, nebo zničením říše) a každá říše se buď stane součástí jiné říše, nebo bude v průběhu dějin zničena.

V provincii je možné civilizovaně žít od jejího osídlení do jejího zničení. Kevina by zajímalo, v jaké provincii se dalo civilizovaně žít co možná nejdéle (kdyby náhodou objevil stroj času a chtěl se někam přestěhovat).

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Formát vstupu: Na prvním řádku vstupu bude počet říší R, počet provincií P a počet událostí N, říše i provincie indexujeme od jedničky. Na dalších N řádcích následují popisy událostí. Na každém řádku s událostí je nejprve rok (celé kladné číslo) a jednopísmenná zkratka typu události. V závislosti na typu události pak mohou následovat další elementy:

- `O` značí osídlení provincie a řádek obsahuje index provincie a index říše.
- `K` značí katastrofu v provincii a řádek obsahuje její index.
- `P` značí podrobení říše a řádek obsahuje index vítězné a index podrobené říše.
- `Z` značí zničení říše a řádek obsahuje její index.

Události nejsou žádným specifickým způsobem seřazeny.

Formát výstupu: Výstupem je číslo provincie, v niž se dá nejdéle civilizovaně žít a délka tohoto období v letech.

Ukázkový vstup:

```
2 3 6
4 K 1
2 0 2 1
5 P 2 1
3 0 3 2
6 Z 2
1 0 1 1
```

Ukázkový výstup:

```
2 4
```

Slibujeme, že v prvních několika vstupech nebudou žádné katastrofy.

33-1-2 Kuchyňská prkénka — 11 bodů

Kevinův strýček truhlář vyrábějící kuchyňská prkénka se jednou rozhodl, že aby ušetřil, tak si koupí jedno dlouhé prkno o délce N centimetrů. To pak chtěl rozřezat na malá prkénka délky D centimetrů, která pak prodává po K korunách.

Koupené velké prkno ovšem obsahuje vady. Vady se sice dají opravit (například zbrousit, zalepit suky a podobně), ovšem opravy nejsou zdarma. Pro každý centimetr proto truhlář určil cenu, za kterou ho lze opravit. Ovšem neví si rady s tím, které části opravit a jak pak velké prkno nařezat na jednotlivá prkénka. Žádá vás tedy o pomoc, jak to udělat, aby měl nejvyšší výdělek.

Na vstupu dostanete N – délku celého prkna, D – délku požadovaných prkének a K – počet peněz, co si vydělá za jedno prkénko. Pak dostanete N přirozených čísel vyjadřujících, za jakou cenu je možné opravit jednotlivé centimetry z prkna (v případě, že je daná část v pořádku, tak je to nula). Prkno je možné řezat pouze v celočíselných vzdálenostech.

Vašim úkolem je vymyslet algoritmus, který pro daný vstup odpoví, jakou největší částku si může truhlář vydělat (prkno má již má koupené, to nepočítejte).

Ukázkový vstup:

```
4 2 5
10 1 2 3
```

Ukázkový výstup:

```
2
```

V tomto případě z prkna vyrobíme jen jedno prkénko délky dva uprostřed (na místech s cenou opravy 1 a 2).

Ukázkový vstup:

```
7 1 7
1 6 8 2 3 7 1
```

Ukázkový výstup:

```
22
```

Tady vyrobíme prkénko délky jedna z každé pozice, kde jde cena za opravu nižší, než 7.

33-1-3 Laser v bludišti — 12 bodů

Ve výzkumném centru LIGO, kde se pomocí laserů měří gravitační vlny, bylo zemětřesení. Všechna zrcadla na odrážení laseru se náhodně pootáčela. Zrcadla jsou ale velmi těžká a křehká, takže je jejich otáčení velmi energeticky náročné. Protože rozpočet na výzkum je omezený granty, vědci musí zjistit jak za co nejméně energie pootáčet zrcadla, aby se laserový paprsek poodrážel ze zdroje do cíle.

Na vědeckém pracovišti se mohou nacházet následující čtyři druhy předmětů:

- `#`: zeď, skrz kterou laser neprosvítí
- `0`–`7`: zrcadlo, které laser odráží
- `Z`: zdroj ze kterého laser svítí
- `C`: cíl, do kterého má laser dosvítit

Číslo na políčku se zrcadlem určuje jeho orientaci. 0 je orientace zrcadlovou plochou nahoru. Pokud číslo o jedna zvětšíme, zrcadlo se otočí o 45 stupňů podle hodinových ručiček. Zrcadlo odráží paprsky pouze z jedné strany, ta druhá je pohlcuje. Otočení zrcadla o libovolný násobek 45 stupňů stojí jednu jednotku energie. Platí, že úhel dopadu se rovná úhlu odrazu. Pokud je plocha zrcadla rovnoběžná s laserem, paprsek zrcadlo mine a políčkem projde jako by to byl vzduch.

Laser ze zdroje vychází vždy směrem nahoru. Skrz zdroj ale laser neprosvítí a zastaví se stejně jako o zeď. Je jedno, ze které strany přijde laser do cíle.

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Formát vstupu: Na prvním řádku vstupu je počet objektů N a rozměry vědecké laboratoře R a S. Na každém z následujících N řádcích je popsán jeden objekt. Nejprve je znak `#`, `Z`, `C` nebo číslo `0` až `7` popisující typ objektu, následují jeho souřadnice X a Y. Osa X směřuje vpravo a Y nahoru. Platí, že $0 \leq X < R$ a $0 \leq Y < S$.

Dále platí, že na každém políčku je nejvýše jeden objekt a na nepopsaných souřadnicích je vzduch, který laser nijak neovlivňuje.

Formát výstupu: Jako výstup vypište, kolik energie řešení stálo a seznam pozic zrcadel, přes které laser prochází (v pořadí od zdroje do cíle).

Ukázkový vstup:

```
20 10 5
4 0 2
Z 1 0
2 1 3
6 1 4
# 3 0
# 3 1
# 3 2
# 3 3
7 4 1
5 4 2
0 4 4
1 6 0
1 6 2
3 6 3
# 8 2
# 8 3
# 8 4
6 9 0
C 9 3
# 9 4
```

Ukázkový výstup:

```
5
1 3
1 4
4 4
4 2
6 2
6 0
9 0
```

Znázornění:

```
.6..0...##
.2.#..3.#C
4..#5.1.#.
...#7.....
.Z.#..1..6
```

33-1-4 Sbírání bonbónů — 12 bodů

Právě stojíme na grafu v jednom z vrcholů. Graf je stromem. Tedy každou dvojici vrcholů spojuje právě jedna posloupnost hran, kde se žádné hrany neopakují. Graf tedy obsahuje N vrcholů a $N - 1$ hran.

V některých vrcholech jsou umístěné bonbóny. Rádi bychom jich nasbírali alespoň K a vrátili se do stejného vrcholu, odkud jsme vyrazili. Bonbónů zvládneme unést libovolné množství a jejich nošení nás nijak neomezuje. Nechceme dělat zbytečně práci navíc. Počet prošlých hran tedy musí být nejmenší možný.

Vaším úkolem je vymyslet a popsat algoritmus, který dostane na vstupu číslo K, strom popsaný seznamem hran mezi vrcholy (s určeným startovním vrcholem) a informaci, ve kterých vrcholech jsou bonbóny. Na výstupu by měl vydat plán vaší cesty po stromě tak, abyste vyrazili ze startovního vrcholu, prošli co možná nejméně hran, skončili opět ve startovním vrcholu a zároveň cestou posbírali alespoň K bonbónů.

33-1-X1 Krycí jména — 10 bodů

Toto je bonusová úloha pro zkušenější řešitele, těžší než ostatní úlohy v této sérii. Nezískáte za ni klasické body, nýbrž dobrý pocit, že jste zdolali něco výjimečného. Kromě toho za správné řešení dostanete speciální odměnu a body se vám započítají do samostatných výsledků KSP-X.

Signor Segreto pracuje pro tajnou službu. Přesněji řečeno je jejím tajemníkem. Každý den se na jeho stole sejdou nejrůznější hlášení od tajných agentů v terénu. Ta je potřeba analyzovat, což obnáší zejména zjistit, které hlášení poslal který agent.

To není jen tak: Nejen že agenti vystupují pod krycími jmény, ale navíc se pod hlášení podepisují zkratkami. A nebohý signor Segreto musí pracně zjišťovat, že pod podpisem „A. Šaf.“ se skrývá agentka Andulka Šafářová, toho času maskovaná jako husopaska.

Vymyslete algoritmus, který bude zkratky luštit. Při spuštění nejprve načte seznam krycích jmen, což jsou dvojice křestního jména s příjmením (dvojice jsou navzájem různé, ale jak křestní jména, tak příjmení se mohou opakovat). Poté bude dostávat dotazy tvořené zkratkou jména a zkratkou příjmení. Na každý dotaz najde krycí jméno, jehož křestní jméno i příjmení začíná zadanými zkratkami. Pokud zkratce žádné jméno nevyhovuje, nebo jich naopak vyhovuje více, vypíše chybovou hlášku.

Uvažujme například následující seznam agentů:

```
Andulka Šafářová
Antonín Šuplíček
Ali Baba
Pěnkavka Šafářová
Popelka Pohádková
```

Ukázkový vstup:	*Ukázkový výstup:*
`A. B.`	`Ali Baba`
`Al. Bab.`	`Ali Baba`
`An. Š.`	`--nejednoznačné--`
`An. Šu.`	`Antonín Šuplíček`
`P. B.`	`--neexistuje--`
`P. Š.`	`Pěnkavka Šafářová`

Poznámka: Očekáváme řešení, které bude na dotazy odpovídat s lepší než lineární časovou složitostí vzhledem k velikosti seznamu jmen.

33-1-S Z hlubin fraktálů — 15 bodů

Letošním ročníkem vás bude provázet seriál. V každé sérii se objeví jeden díl, který bude obsahovat nějaké povídání a navíc úkoly. Za úkoly budete moci získávat body podobně jako za klasické úlohy série. Abyste se mohli zapojit i během ročníku, seriál bude možné odevzdávat i po termínu za snížený počet bodů. Vzorové řešení seriálu proto před koncem celého ročníku KSP uvidí jen ti, kdo už seriál odevzdali.

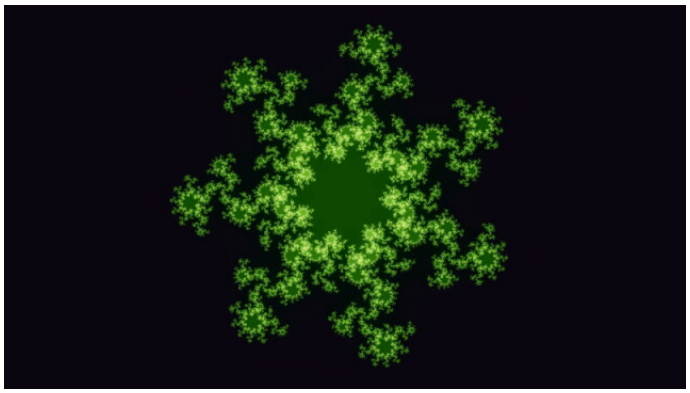
Pojďme si hrát s barvami, náhodnými vzory, paprsky světla a podivnými objekty z hlubin matematiky! Tématem letošního seriálu je počítačová grafika a generování obrazu pomocí shaderů. Cílem je nejen přiblížit vám základní principy vykreslování, ale především vám ukázat, že i z pár řádků kódu můžou vypadnout velmi zajímavé obrázky či animace a že experimentovat s nimi je velká zábava.

Programování nemusí být jen užitečné, může být i **hezké**.

Co bude potřeba

Ze znalostí se vám bude hodit nějaká základoškolská geometrie, konkrétně Pythagorova věta a trigonometrie. Budeme často používat vektory, tedy pokud jste se s nimi ve škole ještě nesetkali, přečtěte si náš krátký úvod do vektorů.[1]

Co se hardwaru týče, potřebujete GPU (a ovladač k němu) podporující WebGL, psát shadery budeme totiž v prohlížeči. Z hlediska výkonu není třeba nic speciálního, seriál je navržen tak, aby šel bez problémů a i s rezervou napsat na Intel HD 630.



Shadery a GPU

Vrhneme se do toho rovnou po hlavě! Shadery jsou krátké prográmky, které umí běžet na **GPU (Graphics processing unit)**. GPU je kus hardwaru uzpůsobený pro akceleraci všemožné grafické práce. Umí například číst z paměti trojúhelníky reprezentované jako trojice souřadnic v prostoru a převádět je na „hotové“ pixely na obrazovce. Některé části tohoto procesu jsou programovatelné právě pomocí shaderů. Shaderů je mnoho druhů, nás ovšem budou zajímat pouze **pixel shadery**, též známé jako **fragment shadery**.

Pixel shadery se provádí na samém konci procesu vykreslování, kdy už víme, které pixely nějaký trojúhelník pokrývá, ale nevíme, jaká bude jejich finální barva – a právě tu spočítá pixel shader. Náš pixel shader se bude spouštět jednou pro každý pixel výsledného obrazu a jeho vstupem budou pouze souřadnice tohoto pixelu.

GPU jsou stroje masivně paralelní. Herní grafické karty mívají tisíce „jader“ (mají blíže k SIMD instrukcím na procesoru než k pravým nezávislým jádrům) a pokud takovou máte, je zcela možné, že se váš shader bude provádět třeba v desetitisíci či více instancích naráz. My si s tímto paralelismem nemusíme dělat těžkou hlavu. Shadery se vždy píší z pohledu jediného vlákna a výpočty různých pixelů se navzájem nemohou nijak ovlivnit (existují vyjímky, my se s nimi ale nesetkáme). Konečný výsledek bude shodný s tím, jako by náš kód běžel ve for smyčce pro každý pixel zvlášť.

[1] `http://ksp.mff.cuni.cz/encyklopedie/vektory.html`

Shadertoy

Vše budeme dělat v Shadertoy,[2] což je webová aplikace, která umožňuje psát pixel shadery a snadno vidět jejich výstup.

Všimněte si trojúhelníkového tlačítka v levém dolním rohu editoru, které shader zkompiluje a začne s ním vykreslovat okno v levé části obrazovky. Pokud máte pomalý stroj nebo náročný shader a seká se vám obraz či prohlížeč, můžete překreslování pozastavit tlačítkem "pause" v dolní části okna s obrazem.

V pravém dolním rohu editoru najdete tlačítko s otazníkem, které zobrazí stručnou nápovědu včetně seznamu zabudovaných funkcí a vstupů.

GLSL

Shadery budeme psát v jazyce **GLSL (OpenGL Shading Language)**, který se dá považovat za zjednodušené a velmi osekané C, takže pokud znáte jiné céčkovské jazyky jako C++, C# nebo Javu, tak v něm budete jako doma. Pokud znáte jen Python, tak se také rychle chytnete, jen nezapoměňte psát za každým řádkem středník ;-)

Rozebereme si jeden jednoduchý shader pro Shadertoy. Je to mírně upravený výchozí shader:

```
vec3 swapRedBlue(vec3 color)
{
    return color.bgr;
}

void mainImage(out vec4 fragColor,
               in vec2 fragCoord)
{
    // Normalizované souřadnice pixelu
    //          (v rozsahu 0..1)
    vec2 uv = fragCoord / iResolution.xy;

    // Barva pixelu měnící se v čase
    vec3 col = 0.5 + 0.5 *
            cos(iTime + uv.xyx + vec3(0, 2, 4));

    col = swapRedBlue(col);

    // Výstup na obrazovku
    fragColor = vec4(col, 1.0);
}
```

Otevřete si Shadertoy a copypastněte do něj tento kód, abyste si se shaderem mohli rovnou hrát. Nezapomeňte jej zkompilovat tlačítkem v levém dolním rohu editoru. Výsledek bude tentokrát téměř totožný s výchozím shaderem v Shadertoy:

[2] `https://www.shadertoy.com/new`

Komentáře v kódu začínají dvěma lomítky. Každý řádek, na kterém je nějaký příkaz, je ukončen středníkem.

Tento shader obsahuje dvě funkce: `swapRedBlue` a `mainImage`. Tělo funkce je blok kódu. Blok kódu bývá obalen složenými závorkami. Před názvem funkce je napsán její návratový typ.

```
vec3 swapRedBlue(vec3 color)
{
    // ...
}

void mainImage(out vec4 fragColor,
               in vec2 fragCoord)
{
    // ...
}
```

Hlavní funkce `mainImage` nevrací nic, což je značeno klíčovým slovem `void`, ale může zapisovat do výstupního parametru `fragColor`, kde je uložena barva (RGBA), která se zobrazí na obrazovku. Dále má jeden vstupní parametr `fragCoord`, což jsou souřadnice středu aktuálního pixelu (v pixelech, tedy tato hodnota je pro levý dolní pixel rovna (0.5, 0.5), pro pixel vpravo od něj (1.5, 0.5) atd.). Každý parametr funkce má též před názvem napsaný svůj typ. Klíčové slovo `in` (před parametrem `fragColor`) je nepovinné, parametr bez něj je automaticky vstupní.

Funkce `swapRedBlue`, která prohodí červený a modrý kanál v nějaké barvě, vrací `vec3` a má jediný vstupní parametr `color`, též typu `vec3`.

Datové typy v GLSL, které budeme používat, jsou tyto:

- `float`: desetinné číslo, skalár
- `vec2`: dvojice floatů či dvojrozměrný vektor, vhodné například pro 2D souřadnice
- `vec3`: trojice floatů, například 3D souřadnice nebo RGB barva
- `vec4`: čtveřice floatů, například RGBA barva
- `int`: celé číslo (může být kladné i záporné)
- `ivec2`: dvojice intů
- `ivec3`: trojice intů
- `ivec4`: čtveřice intů
- `bool`: logická hodnota, true nebo false

Inty a intové vektory se používají málokdy, budeme pracovat především s floaty a jejich vektory. Existují i boolové vektory (např. `bvec2`).

Co ten shader dělá?

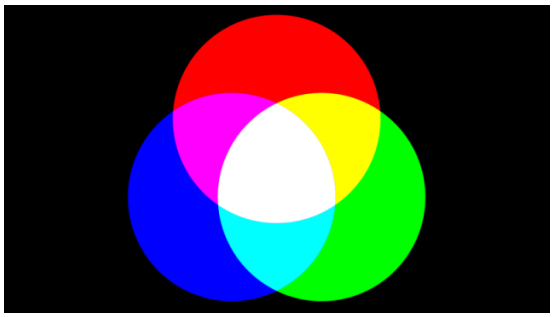
```
void mainImage(out vec4 fragColor,
               in vec2 fragCoord)
{
    // Normalizované souřadnice pixelu
    //          (v rozsahu 0..1)
    vec2 uv = fragCoord / iResolution.xy;

    // Barva pixelu měnící se v čase
    vec3 col = 0.5 + 0.5 *
            cos(iTime + uv.xyx + vec3(0, 2, 4));

    col = swapRedBlue(col);

    // Výstup na obrazovku
    fragColor = vec4(col, 1.0);
}
```

Hlavní funkce nejprve deklaruje proměnnou `uv`, do které hned přiřadí souřadnice pixelu v obrázku, převedené do rozsahu 0 až 1. Hodnota $(0, 0)$ je levý dolní roh obrazu, $(1, 0)$ pravý dolní, $(0.5, 0.5)$ je uprostřed a tak dále. Všimněte si, že je zde počátek souřadnic mírně neintuitivně vlevo **dole**, na rozdíl od běžných grafických programů.

Hodnotu `uv` získá tak, že souřadnice v pixelech vydělí celkovou velikostí obrazu, který je uložená v zabudovaném vstupu `iResolution`. Zabudovaných vstupů je v Shadertoy více, jejich seznam je v nápovědě (otazník v pravém dolním rohu editoru).

Když provádíme nějakou z operací plus, mínus, krát, děleno na dvou vektorech, tak se tato operace provede zvlášť na jednotlivých složkách vektoru. Například `vec2(2, 3) / vec2(4, 5)` nám dá hodnotu `vec2(0.5, 0.6)`. Nepleťte si násobení dvou vektorů po složkách se skalárním a vektorovým součinem, pro ty jsou v GLSL speciální funkce `dot` a `cross`. Aritmetické operace musíme vždy provádět na dvou stejně velkých vektorech nebo na vektoru a skaláru, jak uvidíme později.

Další řádek deklaruje proměnnou `col` a do ní spočítá nějakou zajímavou barvu. Barva je v shaderech tradičně trojice desetinných čísel v rozsahu 0 až 1, které kódují barvu v systému **RGB (red, green, blue)**. Tímto systémem se barvy popisují jako součet (aditivní míchání) různých jasů tří základních barev (červená, zelená, modrá). Například modrá je $(0.0, 0.0, 1.0)$, oranžová je $(1.0, 0.5, 0.0)$. Fungování tohoto barevného systému se nejlépe popíše obrázkem:



Barevný přechod

```
vec3 col = cos(iTime + uv.xyx + vec3(0, 2, 4));
```

Co dělá výraz `cos(iTime + uv.xyx + vec3(0, 2, 4))`?

Asi poznáte, že se v tomto řádku počítá nějaký kosinus. Ale z čeho? Nejdřív se vezme `iTime`, k tomu přičteme `uv.xyx` a k tomu `vec3(0, 2, 4)`.

`iTime` je další zabudovaná hodnota, ve které je ve floatu uložen čas od spuštění stránky v sekundách. Hodí se pro animování různých věcí.

Poslední výraz, `vec3(0, 2, 4)`, vyrobí nový trojrozměrný vektor, kde první hodnota bude 0, druhá 2 a třetí 4. Vektor lze vyrobit například i zápisem `vec3(1.0)`, což vytvoří 3D vektor, jehož všechny složky jsou jedničky, nebo zápisem `vec4(col, 1.0)` (viz poslední řádek hlavní funkce), který vytvoří 4D vektor, jehož první tři složky se překopírují z 3D vektoru `col` a poslední složka bude mít hodnotu jedna.

Prostřední výraz, `uv.xyx`, je zajímavější, protože vytváří `vec3` z proměnné `uv`, která je ale pouze `vec2`. U vektorů můžeme přistupovat k jednotlivým složkám pomocí tečky a písmenka složky, které chceme. Máme-li někde `vec4 v`, tak `v.x` je první složka, `v.y` druhá, `v.z` třetí a `v.w` čtvrtá.

Též můžeme pro přístup k jednotlivým komponentám místo písmen **xyzw** použít písmena **rgba**, jako je tomu ve funkci `swapRedBlue`. Výraz `color.bgr` je tedy ekvivalentní výrazu `color.zyx`. Je vhodné tak učinit, pokud je ve vektoru uložená barva. Dále lze k jednotlivým složkám přistupovat také jako k prvkům pole, například `v[0]` je první složkou vektoru `v`.

Můžeme přistupovat i k více složkám naráz, viz první řádek funkce, kde je `iResolution.xy`, čímž získáme `vec2` z prvních dvou složek `iResolution` (jejíž typ je `vec3`). Nic nám nebrání v tomto zápisu změnit pořadí složek nebo některou použít vícekrát.

Tedy `uv.xyx` je totéž jako zápis `vec3(uv.x, uv.y, uv.x)`.

Nyní víme, že v kosinu sčítáme `iTime`, což je float, s `uv.xyx`, co je `vec3`. Toto není chyba, přestože sčítáme float s trojrozměrným vektorem, protože speciálně aritmetické operace na vektoru a skaláru (jedinému číslu) provádět lze, operace se provede na složkách vektoru zvlášť.

Z toho vyplývá, že `iTime + uv.xyx` je ekvivalentní zápisu `vec3(iTime + uv.x, iTime + uv.y, iTime + uv.x)`.

Též `cos` se provede zvlášť na každé složce vektoru, to samé platí i pro podobné funkce, například absolutní hodnotu `abs`.

Přechody mezi rozsahy

```
col = 0.5 + 0.5 * col;
```

Nyní je v proměnné `col` výsledek kosinu, tedy hodnoty v rozsahu -1 až 1. Nicméně zobrazitelné barvy jsou jen v rozsahu 0 až 1, tedy pokud bychom výsledek kosinu zobrazovali napřímo, přijdeme o onu zápornou půlku rozsahu, která se zobrazí stejně jako hodnota 0.

Proto na tomto řádku přesuneme `col` z rozsahu $-1 \ldots 1$ do rozsahu $0 \ldots 1$.

Prvním problémem je, že pokud si tyto rozsahy představíme na číselné ose, $-1 \ldots 1$ má „šířku“ 2, ale $0 \ldots 1$ má šířku jen 1. Proto `col` vynásobíme jednou polovinou, čímž tuto hodnotu převedeme do rozsahu $-0.5 \ldots 0.5$, který už má správnou šířku. Nyní nám stačí tento rozsah posunout o 0.5 „doprava“, což uděláme přičtením 0.5.

Tento převod mezi rozsahy je v shaderech dost častý a stojí za to si ho pamatovat. Opačný přechod, tedy z rozsahu $0 \ldots 1$ do $-1 \ldots 1$, se provede následovně:

```
value = value * 2.0 - 1.0;
```

Vraťme se k barevnému přechodu. Shader tedy nejprve spočítá kosiny z tří různých hodnot, kde každá je součtem nějaké souřadnice na obrazovce `uv.xyx`, času od počátku `iTime` a konstanty `vec3(0, 2, 4)`, a výsledek namapuje do $0 \ldots 1$. Protože se argument kosinu mění se souřadnicemi pixelu, dostáváme v obrazu barevný přechod z jedné strany na druhou, a protože se mění také v čase, tak se daný přechod postupně posunuje. A protože je každý ze tří argumentů v kosinu posunutý o konstantu (0, 2 nebo 4), získáváme zajímavé barvy.

Schválně zkuste, co se stane bez tohoto posunu! Proč se ve výsledné animaci střídá zelená, černá, bílá a fialová?

Též můžete zkusit vynásobit proměnnou `uv` něčím velkým, třeba 10.0, ještě před řádkem s kosinem. Tím se obraz „odzoomuje“ a budou vidět kosinové vlny jednotlivých barev.

Ještě se vraťme k funkcím. Možná jste zvyklí hojně využívat rekurzi, ale vězte, že GPU žádný zásobník nemají, tudíž

je rekurze v GLSL zakázána. Jinak jsou funkce naprosto v pořádku.

Na posledním řádku hlavní funkce zapíšeme spočítanou barvu do `fragColor`, což je `vec4`, protože výsledkem shaderu je barva ve formátu RGBA, tedy i s alfa kanálem (průhledností). Hodnota alfa kanálu v Shadertoy nemá na nic vliv, je ovšem doporučené nechávat ji nastavenou na 1.

Shrnutí shaderu

Tím jsme prošli celý ukázkový shader. Shrňme si to nejdůležitější, co jsme se naučili:

- každý příkaz končí středníkem, funkce mají vždy deklarovaný svůj návratový typ (na rozdíl třeba od Pythonu)
- barva je trojice desetinných čísel od nuly do jedné
- aritmetika na vektorech funguje po složkách
- funkce typu abs, sin, cos se taky počítají zvlášť po složkách
- k jednotlivým složkám vektoru lze snadno přistupovat stylem `v.x`, `v.zywz`, `v.rgb` atd.
- nefunguje rekurze

Dejte také pozor na to, že když napíšete nějaké celé číslo, například `4`, tak se automaticky považuje za int a překladač si pak občas stěžuje na to, že má nesprávný typ v nějaké funkci. Float z něj uděláte prostě tak, že za něj připíšete desetinnou tečku a nulu: `4.0`.

Případně, pokud potřebujete převést intovou proměnnou na float, dělá se to výrazem `float(value)`. Převod floatu na int se dělá obdobně.

A co když něco nefunguje jak má? Shadertoy neumožňuje debugování, ale lze si poradit i bez něj. Nic vám nebrání si nějakou zajímavou proměnnou vizualizovat pomocí trošky kódu! Zkuste přidat na konec hlavní funkce předchozího shaderu tento řádek: `fragColor = vec4(uv, 0.0, 1.0);`.

Tím přepíšete hodnotu ve `fragColor` na něco, co má v prvních dvou složkách (červená a zelená) hodnotu z `uv`, ve třetí nulu a čtvrté jedničku. To nám krásně vizualizuje právě souřadnice na obrazovce (a opět si všimněte, že $(0, 0)$, tedy černá, je vlevo dole a ne vlevo nahoře).

Mandelbrotova množina

Jedna z nejzajímavějších a zároveň poměrně snadných věcí, které můžeme nakreslit pomocí shaderu, je **Mandelbrotova množina** (též známá jako Mandelbrotův fraktál).

Budeme používat komplexní čísla. Pokud jste se s nimi ještě nesetkali, podívejte se do encyklopedie o vektorech.[3] Pro naše účely je komplexní číslo dvojrozměrný vektor se speciální operací násobení.

Mějme nějaké komplexní číslo c. Definujeme pro něj následující funkci:

$$f_c(z) = z^2 + c.$$

Mandelbrotova množina je množina takových c, pro které je posloupnost $f_c(0), f_c(f_c(0)), f_c(f_c(f_c(0))), \ldots$ omezená, tedy každý její prvek je v absolutní hodnotě menší než nějaké pevné číslo m. V opačném případě by se čísla v posloupnosti v absolutní hodnotě stále více rostla. Za m lze položit číslo 2, protože lze dokázat, že jakmile je nějaký člen posloupnosti v absolutní hodnotě větší než 2, tak tato posloupnost nebude omezená.

V množině jsou tedy ty body komplexní roviny, kde výše popsaná posloupnost nevystřelí do vysokých hodnot. Jaký budou tyto body tvořit tvar? Pojďme si to vizualizovat. Jistě tušíte, že uvidíme něco zajímavého.

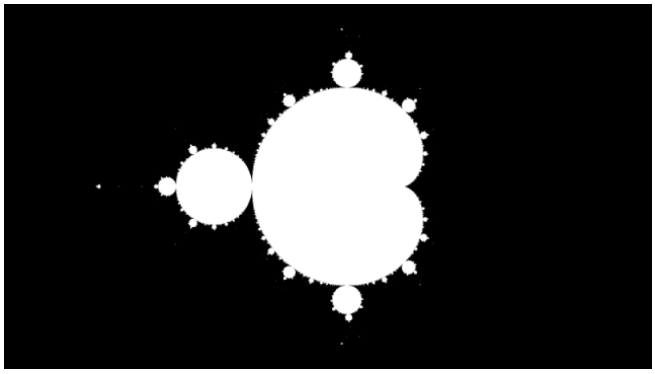
Plochu obrázku namapujeme na komplexní rovinu prostě tak, že x-ovou souřadnici prohlásíme za reálnou část a y-ovou za imaginární. A jak v GLSL reprezentovat komplexní čísla? Bude nám stačit obyčejný `vec2`. Sčítání a odčítání pak funguje jak má (tedy po složkách, pro reálnou a imaginární část zvlášť), jen pro násobení dvou komplexních čísel si potřebujeme napsat vlastní funkci (viz funkce `complexMultiply` níže).

Samozřejmě nemůžeme reálně zkontrolovat celou posloupnost, zda nepřekročí hodnotu 2, ale vystačíme si s tím, že zkontrolujeme prvních 100 prvků ve smyčce `for`.

Množinu vizualizujeme tak, že pokud v ní pixel je (či spíše komplexní číslo, na které jsme jej namapovali), tak jej vybarvíme bíle, pokud v ní není, tak černě. Výsledný kód vypadá takto:

```
vec2 complexMultiply(vec2 a, vec2 b)
{
    return vec2(a.x * b.x - a.y * b.y,
                a.x * b.y + a.y * b.x);
}

bool mandelbrot(vec2 z, vec2 c)
{
    const int iterations = 100;

    for (int i = 0; i < iterations; i++)
    {
        z = complexMultiply(z, z) + c;
        if (length(z) > 2.0)
            return false;
    }

    return true;
}

vec2 mapToComplex(vec2 uv)
{
    vec2 complex = uv * 2.0;
    return complex;
}

void mainImage(out vec4 fragColor,
               in vec2 fragCoord)
{
    vec2 uv = fragCoord/iResolution.xy;

    uv = uv * 2.0 - 1.0;
    // Převedeme uv do rozsahu -1..1

    // "Natáhneme" X tak, aby byl
    // zachován poměr stran
    uv.x *= iResolution.x / iResolution.y;

    // Nyní je tedy čtverec
    // v souřadnicích i čtvercem na monitoru

    vec2 complex = mapToComplex(uv);

    vec3 col = vec3(0.0);

    if (mandelbrot(vec2(0.0), complex))
        col = vec3(1.0);

    fragColor = vec4(col, 1.0);
}
```

[3] `http://ksp.mff.cuni.cz/encyklopedie/vektory.html`



Všimněte si, že ve funkci `mainImage` nějak upravujeme hodnotu `uv`. Samotný první řádek, který jsme používali už dříve, totiž do `uv` uloží souřadnice pixelu v rozsahu od 0 do 1, a to tak, že například nula v x-ové souřadnici znamená levý okraj obrázku a 1 pravý, v y-ové je 0 dolní okraj a 1 horní.

Nicméně zobrazovací plocha je pro vás velmi pravděpodobně obdélníková, takže čtverec v těchto souřadnicích by se na monitoru jevil jako obdélník (o stejném poměru stran jako zobrazovací plocha či monitor).

Toto opravují dva následující řádky. První převede souřadnice do rozsahu -1 až 1. V tomto rozsahu totiž bude platit, že střed obrazu má souřadnice $(0, 0)$, což je velmi užitečná vlastnost.

Druhý přeškáluje x-ovou souřadnici tak, aby byl zachován poměr stran. Nyní -1 či 1 v x-ové ose už nebude odpovídat okrajům zobrazovací plochy, ale pravděpodobně bodům někde těsně před ním (pokud je zobrazovací plocha širší než je vysoká). Díky tomu, že je střed obrazu na $(0, 0)$, s ním škálování nijak nepohnulo a stále se „díváme“ na stejné místo.

Dále si všimněte funkce `mapToComplex`. Souřadnice pixelu nejsou na komplexní číslo převedené přímo, ale jsou změněné tak, aby obraz pokrýval rozsah přibližně $-2 \ldots 2$ a $-2\mathrm{i} \ldots 2\mathrm{i}$ v komplexní rovině. Reálná (x-ová) část bude ve skutečnosti z předchozích operací přeškálována tak, aby byl zachován poměr stran. Bez vynásobení dvěmi v `mapToComplex` bychom viděli jen rozsah přibližně $-1 \ldots 1$ a $-1\mathrm{i} \ldots \mathrm{i}$.

Také ve funkci `mandelbrot` je použita zabudovaná funkce `length`, která vrátí velikost (délku) vektoru, což odpovídá absolutní hodnotě komplexního čísla.

Též jsme zde použili `for` a `if`, které fungují stejně jako v každém jiném céčkovském jazyku.

Nicméně binární černobílý fraktál je docela nuda. Proto funkci fraktálu upravíme tak, aby místo boolu vracela číslo od nuly do jedné. Toto číslo bude nula, pokud c patří do množiny, jinak bude rovno počtu iterací, než prvek posloupnosti překročil absolutní hodnotu vyděleném celkovým počtem iterací.

```
float mandelbrot(vec2 z, vec2 c)
{
    const int iterations = 100;

    for (int i = 0; i < iterations; i++)
    {
        z = complexMultiply(z, z) + c;
        if (length(z) > 2.0)
            return float(i) / float(iterations);
    }

    return 0.0;
}
```

A v hlavní funkci barvu získáme následovně: `vec3 col = vec3(mandelbrot(vec2(0.0), complex));`

Všimněte si, že ve smyčce jsme `i` a `iterations` před dělením převedli na floaty. Jinak by se provedlo celočíselné dělení.

Zkuste si upravený shader spustit. To už je zajímavější, že? Nyní už jsme téměř připravení na nějaké zajímavé úkoly.

Co odevzdávat?

Vždy odevzdávejte zdrojový kód vašich shaderů zabalený v zipku. Sice v tomto díle stačí modifikovat na každý úkol jen několik málo řádků kódu, přesto prosíme odevzdávejte celý shader, který je po překopírování do Shadertoy spustitelný.

Pokud váš shader dělá něco složitějšího, přidejte prosíme ke kódu i slovní komentář.

Pokud naprogramujete například zoom i barevný přechod dohromady v jednom shaderu, můžete jej odevzdat jako jeden soubor. Nemusí platit, že jeden shader je jeden úkol. Pokud tak učiníte, pojmenujte výsledný soubor tak, aby bylo jasné, že se jedná o řešení více úkolů naráz.

Nebudeme hodnotit estetičnost řešení (ale za hezké výsledky budeme rádi :-)), ale funkčnost a zda dělá to, co podle zadání dělat má.

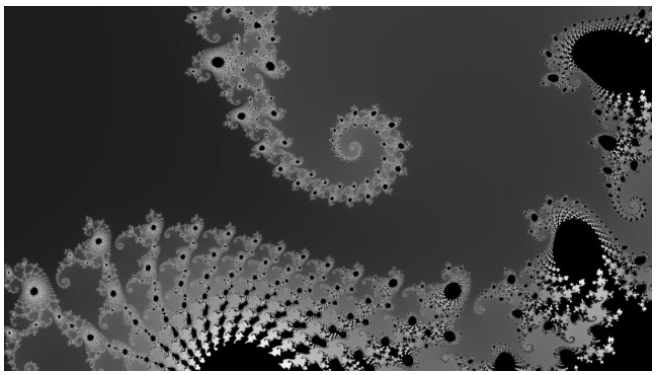
Pokud byste se na něčem zasekli nebo řešili na první pohled nesmyslný bug, neváhejte se nám ozvat. Rádi pomůžeme :-) V shaderech a grafice se občas vyskytují obskurní nástrahy a je lepší nám napsat než si pár dní trhat vlasy. S radostí vám poradíme na mailové adrese *zdrojaky@ksp.mff.cuni.cz*.

Úkol 1 [3b]:

Pojďme si hrát s funkcí `mapToComplex`. Co se stane, když hodnotu proměnné `complex` před vrácení něčím vynásobíte, třeba hodnotou 2.0 nebo 0.5? A co se stane, když k ní něco přičtete? A co když uděláme obojí naráz? Jak se výsledek liší, když nejdřív násobíte a pak přičítáte, a když to uděláme obráceně?

Naprogramujte animovaný zoom na bod v komplexní rovině $-0.745428 - 0.113009\mathrm{i}$. Tento bod by měl být celou dobu uprostřed obrazu, měnit se bude pouze úroveň přiblížení. Inspirujte se některým z předchozích shaderů, který využíval vestavěnou hodnotu `iTime` či nějakou její funkci.

Fraktál zazoomovaný na tento bod by měl vypadat nějak takto:



Můžete si vybrat i jiný zajímavý bod. Buď zkuste najít vlastní nebo si vyberte jeden z tohoto seznamu.[4] Pokud vám tento bod přijde při velkém přiblížení nudný, zkuste zvýšit konstantu `iterations` na něco vyššího, třeba 256.

Pokud při velkém přiblížení vidíte obdélníkové artefakty, tak jste právě narazili na hranici přesnosti 32 bitových floatů. A pokud zvýšíte počet iterací na 2000 a dostatečně zazoomujete, uvidíte další Mandelbrot :-)

Úkol 2 [5b]:

Použijte hodnotu z funkce fraktálu k tomu, abyste vnějšek fraktálu nějak zajímavě obarvili. Místo plynulého přechodu mezi černou a bílou udělejte nějaký zajímavý netriviální barevný přechod. Můžete se inspirovat barevným přechodem z výchozího shaderu. Zároveň ale nechte vnitřek množiny černý!

Můžete použít animaci či zazoomování z minulého úkolu, aby byly vidět detaily fraktálu.

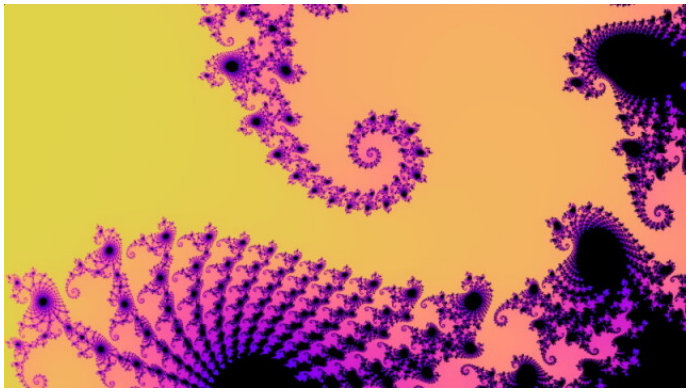
Zde se vám mohou hodit nějaké další zabudované funkce.

První je funkce `clamp(x, a, b)`,[5] která hodnotu x „uzavře“ do rozsahu mezi a a b, pokud v něm už neleží. Všechny parametry musí být stejného typu. Pro vektory zpracovává každou složku zvlášť. Je ekvivalentní zápisu `min(max(x, a), b)`.

Druhou je funkce `mix(x, y, a)`,[6] která vrací přechod mezi hodnotami x a y na základě parametru a. x a y mohou být floaty či vektory (ale vždy musí mít stejný typ).

Funkce se chová tak, že pokud je a nula, tak vrací x, pokud je jedna, tak y, pokud je a 0.5, tak vrací hodnotu na půli cesty mezi x a y a tak dále. Je ekvivalentní zápisu `x * (1 - a) + y * a`. I když hodnota a nemusí být v rozsahu $0 \ldots 1$, většinou chceme, aby tam byla, a právě k tomu se nám hodí `clamp`.

Pro zajímavé přechody můžete použít třeba i funkce `sin` a `cos` nebo funkci `pow(a, b)`, která vrací mocninu a^b. Též mohou být zajímavé jiné barevné soustavy, například *HSV (hue, saturation, value)*, tedy *(odstín, sytost, světlost)*. Kód pro převod z HSV do RGB můžete použít cizí.

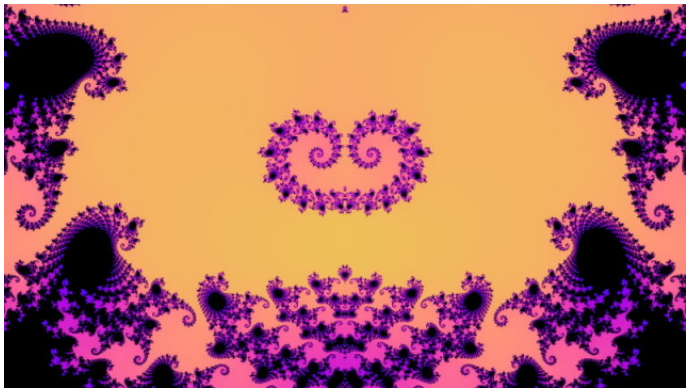


Úkol 3 [2b]:

Změňte funkci `mapToComplex` (ať už původní, nebo vaši vlastní z úkolu 1) tak, aby se obraz zrcadlil podle svislé čáry procházející jeho středem.

Je dobré si uvědomit, že barva každého pixelu opravdu závisí jen na jeho souřadnicích `uv`. Zrcadlení tedy lze dosáhnout tak, že je vhodně zmanipulujeme...

Výsledek by měl vypadat nějak takto:



Úkol 4 [1b]:

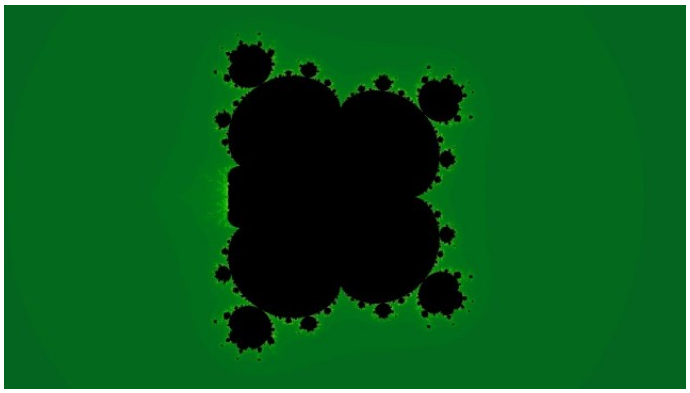
Připomeňme si funkci, ze které fraktál vzniká:

$$f_c(z) = z^2 + c.$$

Používáme zde druhou mocninu. Co kdybychom zkusili použít třetí nebo ještě vyšší? Zkuste to!

Implementujte Mandelbrotův fraktál používající nějakou vyšší celočíselnou komplexní mocninu, třeba z^3. Není třeba, aby byla implementace obecná, stačí, když bude natvrdo zadrátovaná na konkrétní mocninu.

Fraktál také vypadá zajímavě s neceločíselnými mocninami, implementace je ovšem hodně složitá. Pro $f_c(z) = z^{5.5} + c$ vypadá takto:



[4] `http://www.cuug.ab.ca/dewara/mandelbrot/images.html`
[5] `https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/clamp.xhtml`
[6] `https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/mix.xhtml`

Úkol 5 [4b]:

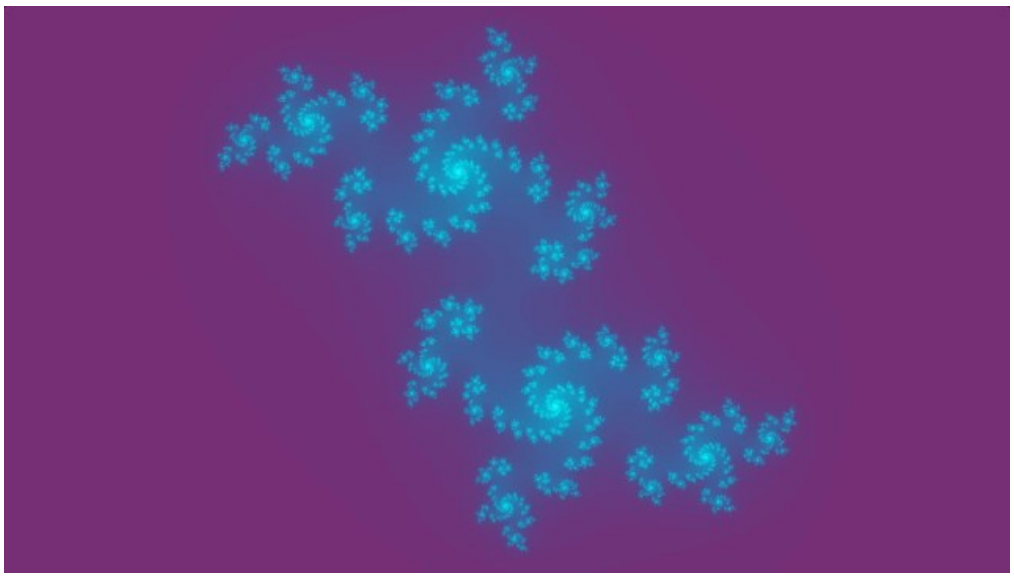
Vraťme se k definici Mandelbrotovy množiny. Počáteční hodnotu jsme zvolili pevně na $0 + 0\mathrm{i}$ a parametr c jsme namapovali na plochu obrazu. Co když to uděláme obráceně, tedy parametr c určíme pevně a na počáteční z namapujeme obraz? Zkuste to! Výsledku se říká **Juliova množina**. Za c zkuste dosadit různé hodnoty. Zajímavé je třeba $0 + 0.7\mathrm{i}$.

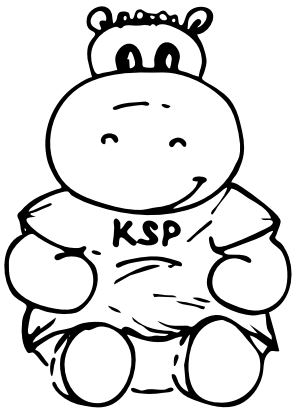
Také ve výsledku animujte parametr c podle času, třeba ho můžete nechat kroužit po jednotkové kružnici. Použijte barevný gradient z předchozích úkolů. Odevzdejte zdrojový kód shaderu s touto animací.

Kam dál?

Pokud vás shadery nadchly a chcete si s nimi hrát nezávisle na seriálu, mohl by vás zajímat The Book of Shaders.[7] Je to velmi hezky zpracovaná webová učebnice shaderů, ve které najdete také různé šumové funkce, vzory a co se s nimi dá všechno dělat. Tento seriál je jí inspirovaný a je na ní částečně založený. Vězte, že tam najdete spoilery na příští díl, který se bude věnovat šumovým funkcím :-)

Kuba Pelc





[7] `https://thebookofshaders.com/`

Recepty z programátorské kuchařky: Základní algoritmy

Tato naše kuchařka je nejzákladnější ze základních a je určena hlavně pro začínající řešitele. To však neznamená, že zkušenější řešitelé do ní nahlédnout nemůžou – třeba na nějakou konkrétní programátorskou techniku, kterou by si potřebovali osvěžit.

V první části kuchařky se seznámíme hlavně se základními principy programování, uchovávání dat v počítači a základy rychlé manipulace s nimi. Po přečtení této části bychom měli být schopní převést své myšlenky z hlavy na papír či do počítače a měli bychom vědět, proč je námi zvolený postup rozumný.

Druhá část nás poté seznámí se základními postupy, jak řešit určité konkrétní problémy. Naučíme se například, jak rychle vyhledávat v uspořádané posloupnosti hodnot nebo jak si pomocí předpočítání usnadnit řešení těžké úlohy.

Většinu klíčových částí se pokusíme též ukazovat v podobě zdrojového kódu ve dvou různých jazycích (v nízkoúrovňovém C, kde je zápis blízký tomu, jak počítač doopravdy pracuje, a v Pythonu, ve kterém se píše o něco příjemněji). Nebudeme ale probírat základy syntaxe těchto jazyků, ty si případně můžete nastudovat jinde.[8]

Část první: Základní pojmy

Algoritmus a program

Pod tajemným slovem *algoritmus* se skrývá jen jiný výraz pro postup. Můžete si to představit jako příkaz od maminky „Běž do krámu, kup chleba, a když budou mít měkké rohlíky, tak jich vem tucet“.[9]

Takovýto příkaz klidně můžeme nazvat algoritmem, ačkoliv to bude asi znít nezvykle – pojem algoritmus se totiž používá hlavně ve světě počítačů. Je to tedy nějaká posloupnost základních příkazů, která řeší nějaký problém. Výběr konkrétního programovacího jazyka rozhoduje o tom, jaké základní příkazy budeme mít k dispozici. Většinou jsou ale skoro stejné.

Mezi základní příkazy patří:

- Manipulace s daty v paměti (uložení či načtení hodnoty, detailněji v další kapitole).
- Provedení nějakého numerického výpočtu ($+, -, *, /$).
- Vyhodnocení nějaké konkrétní podmínky a odpovídající větvení programu: *Pokud platí A, tak proveď B, jinak proveď C.* Přitom B i C mohou být klidně celé *bloky kódu*, tedy libovolně mnoho dalších základních příkazů.
- Opakování nějakého příkazu: *Dokud platí A, dělej B.* Takové konstrukci říkáme *cyklus* a podobně jako u podmínky může být B blok kódu, který se celý opakuje.
- Vstup a výstup programu (typicky vstup od uživatele z klávesnice či načtení vstupu ze souboru; výstup pak může znamenat vypsání výsledku na obrazovku nebo třeba zapsání dat do souboru).

Z těchto základních stavebních kamenů se skládá každý algoritmus. Programem potom rozumíme realizaci algoritmu v nějakém konkrétním programovacím jazyce.

U složitějších programů se pak často setkáme s problémem, že budete mít nějakou posloupnost příkazů, která se bude na spoustě míst programu opakovat, což zbytečně prodlužuje a znepřehledňuje kód.

Řešením tohoto problému je použití *funkcí*. Funkci si můžeme představit jako nějakou pojmenovanou část programu (s vlastní pamětí), kterou můžeme opakovaně použít tím, že ji v různých částech programu *zavoláme*. Funkci při zavolání předáme parametry (například seznam čísel), které se dostanou do její vnitřní paměti.

Funkce pak na základě obdržených parametrů může provádět nějaké operace, při kterých pracuje se svojí vnitřní pamětí (mluvíme o *lokální* paměti, změny v ní se neprojeví nikde mimo funkci). Na konci nám funkce může vrátit nějaký výsledek. Pokud funkce během svého běhu změní i nějaká data v *globální* paměti, či provede nějakou globální operaci (například výpis textu na monitor), mluvíme pak o funkci s *vedlejšími efekty* (neboli side-efekty).

Konkrétním příkladem může být funkce, která nám spočítá odmocninu ze zadaného čísla. Ta dostane jako svůj parametr číslo, uvnitř si provede nějaký výpočet, o který se jako uživatel funkce nemusíme starat, a jako výstup nám vrátí spočtenou odmocninu.

Reprezentace dat v počítači

Celkem často si v průběhu výpočtu našeho algoritmu potřebujeme pamatovat nějaké hodnoty. K tomu nám programovací jazyky dávají nástroj s názvem *proměnná*. Ta představuje jakési pojmenované místo v paměti (přihrádku), do kterého si můžeme data ukládat a pak je odtud zase načítat.

Typickým příkladem může být počítání součtu čísel, která nám uživatel zadá na vstupu. Na začátku nejdříve do nějakého místa v paměti uložíme hodnotu 0. Poté postupně, jak nám uživatel zadává čísla, tuto proměnnou pokaždé přečteme, k její hodnotě přičteme nově zadané číslo a výsledek opět uložíme na stejné místo.

Takovéto použití jedné proměnné je velmi jednoduché (tak jednoduché, že ho takto podrobně do řešení KSPčka nepište, není to potřeba), ale také celkem omezené. Co kdybychom si chtěli pamatovat třeba celou zadanou posloupnost čísel? Mohlo by nám stačit vyrobit si spoustu různě pojmenovaných proměnných, ale nejde to lépe? Jde.

Jednotlivé proměnné se mohou kombinovat do složitějších konstrukcí, které obecně nazýváme *datovými strukturami*. Zkusíme si ty nejzákladnější představit.

Pole

První datovou strukturou, kterou si představíme a která se na výše nastíněnou situaci náramně hodí, je *pole*. To představuje spoustu přihrádek (proměnných) naskládaných v paměti za sebou, ke kterým typicky přistupujeme přes jeden společný název pole a jejich pořadové číslo neboli index (jako `NazevPole[0], NazevPole[1], ...`).[10]

Ve většině základních jazyků je pole jen *statické*, tedy v okamžiku jeho vytváření musíme počítači říct, jak ho chceme

[8] `http://ksp.mff.cuni.cz/study/odkazy.html`

[9] A jako slušně vychovaní se tedy vydáte do krámu a koupíte tucet chlebů, protože měli měkké rohlíky :-)

[10] Pozor, ve světě počítačů se velmi často indexuje od nuly, tedy první prvek má v tomto případě index 0.

velké. Některé vyšší jazyky ale nabízejí i pole, které se dynamicky zvětšuje, takovou konstrukci si ukážeme ve druhé části kuchařky.

Abychom nebyli omezeni jen jedním rozměrem, můžeme si klidně vyrobit pole dvourozměrné (případně obecně n-rozměrné). Dvourozměrné pole je vlastně tabulka hodnot, nazýváme ji také někdy *matice*, a může se nám hodit například při reprezentaci různých map (plán bludiště) nebo, jak uvidíme níže, pro reprezentaci dalších datových struktur.

U pole již má smysl přemýšlet, jak dlouho bude která operace trvat. Díky tomu, že jsou jednotlivé prvky v poli naskládané pevně za sebou, je snadné spočítat umístění konkrétní přihrádky. Proto když se počítače zeptáme na obsah přihrády `pole[42]`, vrátí nám hodnotu ihned.

Tomu budeme říkat *operace v konstantním čase* a budeme značit, že trvá čas $\mathcal{O}(1)$. Efektivitu programu totiž nepočítáme v sekundách (protože každý z nás má asi jinak rychlý počítač), ale v počtu základních operací, které musí program řádově vykonat. Více o časové složitosti si můžete přečíst v kuchařce o složitosti,[11] nejdříve však doporučujeme dočíst tuto kuchařku.

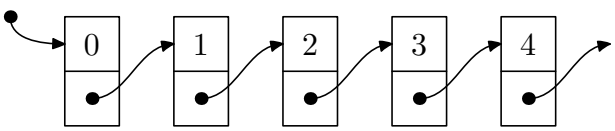
Přidání nového prvku na konec pole také zvládneme v konstantním čase. Problém je přidání nového prvku někam doprostřed (což se nám typicky stane, pokud budeme chtít udržovat hodnoty v poli seřazené a zároveň do něj vkládat nové). V takovém případě se totiž všechny prvky za vkládaným musí posunout o jednu pozici dál, aby se vkládaný prvek vešel na své místo. Taková operace tedy může pro pole délky N (čili pole obsahující N prvků) trvat řádově až N kroků, což zapisujeme jako $\mathcal{O}(N)$ a říkáme, že je to vzhledem k N *lineární časová složitost*.

To je značná nevýhoda oproti struktuře, kterou si ukážeme za chvíli. Určitě ale pole nezavrhujme. Je to základní datová struktura, která nalezne použití ve spoustě programů, a jak si ve druhé části kuchařky ukážeme, můžeme ho použít třeba k rychlému hledání hodnoty metodou *binárního vyhledávání*. Nyní ale již slibovaná další datová struktura.

Spojový seznam a ukazatele

Pole jsme měli v paměti určené jenom tím, že počítač věděl, kde je jeho začátek a kolik místa v paměti zabírají jeho prvky. Při dotazování na konkrétní index pak podle indexu a podle velikosti prvků počítač přesně věděl, kam do paměti se má podívat, aby našel námi požadovaný prvek (to vše zvládl v konstantním čase). Jednotlivé prvky si tedy vůbec nemusely pamatovat, kde se nachází jejich sousedi, protože všechny prvky seděly v paměti za sebou.

Představme si ale teď situaci, kdy by si každý prvek ještě pamatoval pozice sousedů. Pak bychom mohli mít prvky libovolně rozházené v paměti a jen by se na sebe vzájemně odkazovaly (první prvek by tvrdil, že druhý je na pozici X, druhý by tvrdil, že třetí je na pozici Y, a tak dále).

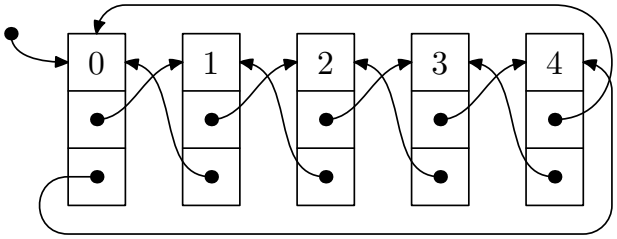


K lepšímu pochopení tohoto principu je důležité si vysvětlit, co to je *ukazatel* (nebo také *odkaz* či anglicky *pointer*). Každé místo v paměti počítače má své číselné označení, kterému říkáme *adresa*. Když si vytváříme nějakou pojmenovanou proměnnou, ta se vlastně odkazuje na určité místo v paměti a na tomto místě v paměti je její hodnota.

Co kdyby ale hodnota proměnné byla adresa nějakého jiného místa v paměti? Pak takové proměnné říkáme *pointer* a umožňuje nám vytvářet výše popsanou strukturu rozházených prvků v paměti.

Spojový seznam je tedy určený svým prvním prvkem (máme v jedné proměnné pointer na tento prvek, který se často nazývá *kořen*, protože z něj „vyrůstá“ zbytek struktury) a poté u každého dalšího prvku máme za sebou uloženou hodnotu tohoto prvku a odkaz (pointer) na další prvek. Odkazy mezi prvky mohou být i obousměrné, mohou vést dokola (poslední ukazuje na první) či mohou dokonce tvořit nějakou složitější strukturu (pak to ale již nebude čistý spojový seznam).

Pokud pointer nemá nikam ukazovat, realizuje se to odkázáním tohoto pointeru na adresu `NULL`. To skoro doslovně říká „Neukazuji nikam“.



Co nám takto vystavěná struktura umožňuje v porovnání s polem? Přístup na konkrétní prvek v ní stojí lineárně času, protože ho musíme „odkrokovat“ od prvního prvku (na který máme pointer), tedy musíme udělat až $\mathcal{O}(N)$ kroků. Pokud bychom však pointer na daný prvek už nějak měli, samozřejmě na něj můžeme přistoupit v konstantním čase.

Naopak přidávání prvků na konkrétní místo (i jejich odebírání) máme v podstatě zadarmo a spojový seznam můžeme rozšiřovat, dokud na něj máme v počítači paměť. Ve chvíli, kdy chceme přidat nový prvek za prvek, na který máme pointer, jen šikovně přepojíme ukazatele. Pokud předtím ukazatele vedly $A \to B$, teď povedou $A \to C \to B$ (a při odebírání naopak).

Zde můžete vidět ukázku pointerů a spojových seznamů v jazyce C, kde jsou tyto věci mnohem více nízkoúrovňové (ale zato rychlejší):

```
#include <stdio.h>
#include <stdlib.h>
// Příkazy výše načetly do programu
// standardní knihovny a funkce z nich.

// Struktura pro prvek obsahující dopředné
// i zpětné odkazy. Zkráceně tomuto typu
// budeme říkat "tprvek".
typedef struct prvek tprvek;
struct prvek {
    int hodnota;
    tprvek *dalsi;
    tprvek *predchozi;
};

// Vytvoří nový prvek:
tprvek *novy(int i) {
    tprvek *aktualni =
```

[11] `http://ksp.mff.cuni.cz/viz/kucharky/slozitost`

```
        malloc(sizeof(tprvek));
    aktualni->dalsi = NULL;
    aktualni->predchozi = NULL;
    aktualni->hodnota = i;
    return aktualni;
}

// Odstraní prvek a vrátí pointer na další
// prvek (vrácení pointeru se hodí při
// odstraňování kořene):
tprvek *odstran(tprvek *aktualni) {
    if (aktualni->predchozi != NULL)
        aktualni->predchozi->dalsi =
            aktualni->dalsi;
    if (aktualni->dalsi != NULL)
        aktualni->dalsi->predchozi =
            aktualni->predchozi;

    tprvek *pomocna = aktualni->dalsi;
    free(aktualni);
    return pomocna;
}

// Vloží a vrátí pointer na nový prvek:
tprvek *vloz_za(tprvek *aktualni, int i) {
    tprvek *pomocna = aktualni->dalsi;
    aktualni->dalsi = novy(i);

    if (pomocna != NULL)
       pomocna->predchozi = aktualni->dalsi;

    aktualni->dalsi->dalsi = pomocna;
    return aktualni->dalsi;
}

// Použití:
int main(void) {
    tprvek *koren = novy(1);
    tprvek *aktualni = vloz_za(koren, 2);

    aktualni = koren;
    while (aktualni != NULL) {
        printf("%d\n", aktualni->hodnota);
        aktualni = aktualni->dalsi;
    }

    return 0;
}
```

Zde je ukázka spojových seznamů v Pythonu, kdybychom si je podobně jako v C chtěli naprogramovat sami (Python totiž obsahuje spoustu základních struktur již hotových, podívejte se na modul jménem `collections`):

```
class Prvek:
    def __init__(self, hodnota):
        self.hodnota = hodnota
        self.dalsi = None
        self.predchozi = None

class Spojak:
    def __init__(self):
        self.koren = None

    def vypis(self, aktualni):
        if aktualni is not None:
            print(aktualni.hodnota)
            self.vypis(aktualni.dalsi)

    def vloz_po(self, prvek, za_prvek = None):
        if za_prvek is not None:
            prvek.dalsi = za_prvek.dalsi
            prvek.predchozi = za_prvek
            za_prvek.dalsi = prvek

        if prvek.dalsi is not None:
            prvek.dalsi.predchozi = prvek

        if self.koren is None:
            self.koren = prvek

    def odstran(self, prvek):
        if prvek.predchozi is not None:
            prvek.predchozi.dalsi = \
                prvek.dalsi
        if prvek.dalsi is not None:
            prvek.dalsi.predchozi = \
                prvek.predchozi

# Použití:
prvekA = Prvek("A")
prvekB = Prvek("B")
prvekC = Prvek("C")
prvekD = Prvek("D")

seznam = Spojak()

seznam.vloz_po(prvekB)
seznam.vloz_po(prvekD, prvekB)
seznam.vloz_po(prvekC, prvekD)
seznam.vloz_po(prvekA, prvekC)
seznam.odstran(prvekC)

seznam.vypis(seznam.koren)
```

Fronta a zásobník

S použitím spojových seznamů (nebo v jednodušším případě dokonce i polí) můžeme zkonstruovat dvě velmi užitečné datové struktury, frontu a zásobník.

Fronta funguje tak, jak si ji asi každý z nás představuje: ten, kdo se do fronty postaví první, také první přijde na řadu. Trochu jinak si ji můžeme představit jako trubku, do které na jedné straně sypeme nějaké věci a na druhé je odebíráme. Anglicky je též nazývaná *FIFO* (*„First In, First Out“*).

Praktickou realizaci uděláme jednoduše spojovým seznamem. Budeme si držet dva ukazatele, jeden na začátek seznamu, druhý na konec. Když se objeví nový prvek, který do fronty budeme chtít vložit, přidáme ho na konec, zatímco při odebírání z fronty využijeme druhého ukazatele a vezmeme prvek ze začátku.

Druhou velmi podobnou datovou strukturou je *zásobník*. Jak už ale plyne z anglického názvu *LIFO* (*„Last In, First Out“*), funguje spíše jako plný šuplík: Nahoru do něj přidáváme nové prvky, a když chceme nějaký odebrat, vezmeme také zvrchu. To znamená, že první se na řadu dostane naposledy vložený prvek.

Implementace je velmi obdobná jako u fronty, jen bude ukazatel pouze jeden a bude ukazovat jenom na jeden konec spojového seznamu, konkrétně na poslední prvek.

Knihovny

Tyto základní struktury už jsou často předpřipravené jako součást určitých *knihoven* v daném jazyce. Knihovna je většinou sbírka nějakých navzájem souvisejících funkcí, které již někdo sepsal a které si můžeme do našeho programu načíst a používat. Ukázku načtení knihoven můžete vidět například ve výše zmíněném kódu v jazyce C.

Je ale velmi důležité rozumět tomu, jak knihovní funkce

vnitřně fungují. Protože jedině když budeme vědět, co je jak rychlé a efektivní, budeme schopni psát rychlé programy.

Teď již víme, jak reprezentovat nejzákladnější datové struktury v počítači, ale mohlo by se nám hodit zastavit se ještě chvíli u dalších struktur. Tentokrát je už budeme studovat trochu teoretičtěji.

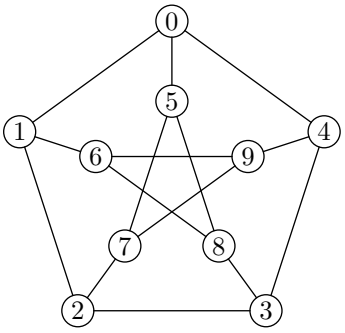
Stromy a grafy v informatice

Grafy

S nějakými grafy jste se již možná potkali, ale tento pojem je bohužel docela přetěžovaný. Jedním jeho významem jsou „koláčové grafy“ a jiné další diagramy znázorňující nějaký poměr (ať už to jsou výsledky voleb, nebo poměr lidí, kteří sledovali v televizi Večerníček).

Další význam můžeme nalézt v analytické matematice, kde se potkáme s grafy průběhu nějakých funkcí. My však nemáme na mysli ani jedno z výše zmíněných, teď se budeme bavit o *kombinatorických grafech*.

Grafem tedy máme na mysli nějakou množinu objektů, říkejme jim *vrcholy*, a nějaké vztahy mezi nimi. Tyto vztahy nazýváme *hranami* a jsou vyjádřené dvojicemi vrcholů, mezi kterými vedou. Ukázku takového grafu vidíme třeba na následujícím obrázku.



Jako praktickou ukázku grafu si můžeme například představit silniční síť nějakého státu: vrcholy budou města a hrany budou silnice, které mezi nimi vedou.

Občas se můžete setkat s pojmem *souvislý* graf. Ten znamená jen to, že mezi každými dvěma vrcholy existuje nějaká cesta. Pokud tomu tak není, je graf *nesouvislý* a dá se rozložit na několik menších grafů, které již souvislé jsou a říká se jim *komponenty souvislosti*.

Samotný graf poté můžeme doplnit tím, že si v každém vrcholu nebo na každé hraně budeme pamatovat nějakou hodnotu (například cenu nejlevnějšího benzínu ve městech a délku v kilometrech na silnicích). Pamatování si hodnot ve vrcholech je docela obvyklá technika a nemá speciální název, ale pokud budeme mít graf, který si pamatuje hodnoty na hranách, budeme o něm mluvit jako o *ohodnoceném grafu*.

Další možnou úpravou je, že každá hrana povede jen jedním směrem (jednosměrné silnice), takovým grafům říkáme *orientované* (pokud pak v orientovaném grafu chceme silnici oběma směry, prostě do něj přidáme dvě hrany, jednu v každém směru).

Poslední, co nám schází k praktickému použití grafů, je naučit se, jak je reprezentovat v počítači. Existuje několik možností (v popisech bude n značit počet vrcholů, m počet hran):

- **Seznam sousedů** – vrcholy grafu budeme mít uložené v poli a u každého vrcholu budeme mít (spojový) seznam čísel dalších vrcholů, do kterých z aktuálního vrcholu vede hrana. Zabírá místo $\mathcal{O}(n+m)$ a hodí se pro řídké grafy (tedy grafy, kde je m řádově stejné jako n).
- **Matice sousednosti** – tabulka $n \times n$, kde na souřadnicích $[i, j]$ je jednička (nebo jiná hodnota, v případě ohodnoceného grafu), pokud z i do j vede hrana, a nula, pokud tam hrana není (u neorientovaných grafů je navíc matice symetrická – je jedno, jestli vezmeme $[i, j]$ nebo $[j, i]$). Hodí se pro husté grafy, kde $m \sim n^2$.
- **Matice incidence** – řádky reprezentují vrcholy, sloupce hrany. V každém sloupci jsou právě dvě jedničky – indexy vrcholů, mezi kterými hrana vede. Zabírá však $\mathcal{O}(mn)$ a její použití bývá dost neohrabané, takže je většinou lepší dát přednost jiné reprezentaci grafu. Je ale dobré o ní vědět.

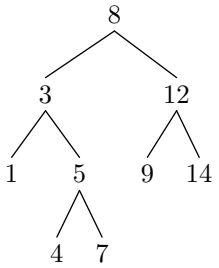
Grafy jsou velmi široké téma. Můžeme hledat jejich minimální kostry, můžeme v nich hledat nejkratší cesty či skrze ně pouštět pod tlakem vodu. Více o nich si tedy můžete přečíst v některé z našich specializovaných grafových kuchařek, které odkazujeme z našeho kuchařkového rozcestníku.[12]

Stromy

Možná si říkáte, co má informatika u všech elektronů společného s lesnictvím? Kupodivu celkem mnoho a bez stromů bychom se v leckterém případě jen těžko obešli. Informatické stromy sice nejsou většinou tak zelené, mají ale, na rozdíl od svých dřevnatých sourozenců, mnoho jiných pěkných vlastností.

Strom je vlastně speciálním případem souvislého grafu, který neobsahuje žádnou kružnici (cyklus). To znamená, že mezi každými dvěma vrcholy stromu existuje právě jedna cesta.

Díky této vlastnosti můžeme nějaký zvolený vrchol prohlásit za *kořen* a strom za něj pomyslně zavěsit (tak, že strom roste od kořene směrem dolů), této operaci se říká *zakořenění*. Pak můžeme mluvit o tom, že z kořene směrem dolů (informatické stromy mají tradičně kořen nahoře) vyrůstají nějaké *podstromy*.



Pokud je strom zakořeněný, můžeme v něm mluvit o *hloubce* každého vrcholu, neboli o jeho vzdálenosti od kořene. Hloubka celého stromu je pak nejdelší ze vzdáleností od kořene k nějakému z *listů* (tak říkáme vrcholům, které již nemají žádné *syny*, tedy vrcholy, které by z nich vyrůstaly). Podle hloubky poté můžeme vrcholy stromu uspořádat do jednotlivých *hladin*.

Velmi často používáme stromy, které jsou nějak pravidelné. Příkladem jsou třeba *binární stromy*, které mají v každém vrcholu maximálně dva syny (říkáme jim *levý a pravý podstrom*). Reprezentovat se dají buď obecně jako každý jiný

[12] `http://ksp.mff.cuni.cz/kucharky/`

strom (v každém vrcholu spojový seznam podstromů), nebo velmi pěkně i v poli.

Stačí si pomyslně doplnit binární strom na *úplný* (to je takový, který má všechny své hladiny plné) a pak ho od kořene směrem dolů po hladinách očíslovat (kořen dostane číslo nula, jeho synové čísla jedna a dva, další hladina čísla tři až šest, atd.).

Můžeme si všimnout, že pokud si v takovém očíslování vezmeme jakýkoliv vrchol s číslem (indexem) i, jeho synové jsou právě vrcholy s indexy $2i+1$ a $2i+2$. Do pole níže je zapsaný binární strom z obrázku výše.

index	0	1	2	3	4	5	6	7	8	9	10
hodnota	8	3	12	1	5	9	14	–	–	4	7

Jak plyne z očíslování, pro úplný binární strom je uložení v poli efektivní a neplýtváme místem. Pokud ale strom úplný nebude, zůstanou nám v poli volná místa. Uložení v poli se tedy vyplatí jen pro stromy, které se od úplných příliš neliší.

Speciálním případem binárních stromů jsou pak ještě *binární vyhledávací stromy*. Jsou to normální binární stromy, pro něž navíc platí, že ať si vezmeme libovolný vrchol, budou hodnoty vrcholů v jeho levém podstromu menší než hodnota tohoto vrcholu a hodnoty v jeho pravém podstromu naopak větší.

V takovém stromu pak zvládneme snadno vyhledávat. Budeme ho postupně procházet od kořene a v jednotlivých vrcholech budeme porovnávat hledanou a aktuální hodnotu a podle toho sestupovat do správného podstromu. Podobná technika je detailněji popsaná ve druhé části kuchařky, v sekci *Rozděl a panuj*.

Na složitější datové struktury stavějící na těchto základních (haldy, intervalové stromy, ...) se můžete podívat do některé z našich dalších kuchařek, na jejichž přehled jsme vás už odkázali o kapitolu výše.

Část druhá: Programátorské techniky

Tato část by měla sloužit jako rychlý přehled a ukázka různých technik, které se dají použít při řešení úloh z KSPčka, nebo při programování obecně.

Rekurze

Rekurze je velmi důležitá programátorská technika. V podstatě znamená definování nějaké věci (ať už je to nějaký objekt či postup výpočtu) pomocí sebe sama.

Rekurzivně může být například zadána nějaká datová struktura. Třeba stromy jsou pěkným příkladem rekurzivně definované datové struktury – každý vrchol stromu může mít syny, a každý z těchto synů je sám o sobě strom (tedy i osamocený vrchol bez synů je stromem).

Prakticky je to realizováno tak, že každý vrchol má svou hodnotu a pak ještě seznam ukazatelů vedoucích na další případné podstromy. S ukazateli jsme se již potkali a s jejich pomocí jsme si postavili spojový seznam. A přesně tak, spojový seznam je také ve své podstatě rekurzivní datová struktura.

Kromě rekurzivních datových struktur se ale často potkáváme i s rekurzivním postupem výpočtu nějakého programu, nejčastěji realizovaným ve formě funkce, která volá sama sebe (většinou s jinými parametry, jinak by to asi nemělo smysl), takové funkci se říká *rekurzivní funkce*.

U rekurzivních funkcí je nejdůležitější věc definovat nějakou *koncovou podmínku*, tedy podmínku, při níž už se rekurze zastaví. Jinak by se totiž mohlo stát, že by rekurze běžela donekonečna.

Přesněji rekurze by se i tak v nějakou chvíli zastavila, ale skončila by chybou, protože by jí došla paměť – každé volání funkce si totiž ukousne kus paměti (musí si pamatovat, kam se po skončení má vrátit), a pokud má rekurzivní funkce ještě nějaké lokální proměnné, musíme si někde uložit všechny lokální proměnné funkcí, z kterých jsme se doposud nevrátili.

Rekurzivní funkce a převod na nerekurzivní cyklus

Typickým příkladem rekurzivní funkce je výpočet Fibonacciho čísel. Ta jsou definována tak, že $f_0 = 1$, $f_1 = 1$ a n-té Fibonacciho číslo je součtem dvou předchozích ($f_n = f_{n-1} + f_{n-2}$). To nám dává posloupnost čísel 0, 1, 1, 2, 3, 5, 8, 13, ... pokračující donekonečna. Pokud toto přepíšeme do programového kódu, tak dostáváme následující zápisy:

V jazyce C:

```
int fib(int n) {
        if (n==0) return 0;
        else if (n==1) return 1;
        else return fib(n-1) + fib(n-2);
}
```

V Pythonu:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Jak vidíme, je přepis celkem přímočarý. Pokud by se nám však rekurze v nějakém případě nelíbila, můžeme se každé rekurze zbavit. Rekurzivní volání totiž můžeme šikovně přepsat na nějaký cyklus se *zásobníkem*.

Pak jen v cyklu odebíráme prvky ze zásobníku, dokud není prázdný, a za každé rekurzivní zavolání do zásobníku přidáme parametry, se kterými bychom naši funkci volali. Tímto postupem převedeme každou rekurzivní funkci na nerekurzivní.

Ještě doplníme poznámku, že ve většině programovacích jazyků každé volání funkce stojí nějaký čas, sice malý, ale když se volání provádí opakovaně, tak se to už nasčítá. Pro reálnou implementaci je tedy nejlepší pokusit se rekurzi převést na nerekurzivní volání, pokud to nějak rozumně jde.

Občas to jde dokonce i jednodušeji a bez zásobníku. Podívejte se na alternativní variantu výpočtu Fibonacciho čísel níže a rozmyslete si, co dělá.

V jazyce C:

```
int fib2(int n) {
        if (n == 0) return 0;

        int a = 0; int b = 1;
        while (n > 1) {
                int pomocna = a + b;
                a = b;
                b = pomocna;
                n--;
        }
```

```
        return b;
}
```

V Pythonu:

```
def fib2(n):
    if n == 0:
        return 0
    a = 0; b = 1
    while n > 1:
        (a, b) = (b, a+b)
        n -= 1
    return b
```

Jak vidíte, je i tato funkce elegantní a navíc běhá mnohem rychleji než její rekurzivní varianta. Tato funkce běhá v $\mathcal{O}(n)$, kdežto rekurzivní varianta počítala stejné věci mnohokrát dokola (zkuste si nakreslit nějaký strom volání předchozí funkce, případně se podívat dopředu do podkapitolky Předpočítané mezivýsledky).

Rekurzivní varianta v tomto případě může běžet až v čase $\mathcal{O}(2^n)$, což je pro velká n mnohem pomaleji než $\mathcal{O}(n)$. Dá se ale celkem lehce rozmyslet úprava rekurzivní varianty, aby běžele také v $\mathcal{O}(n)$, zkuste si rozmyslet jak.

Backtracking

S rekurzí silně souvisí i pojem *backtracking*, česky by se snad dalo říci „metoda pokusu a omylu“. Tímto pojmem označujeme proces, kdy postupně zkoušíme všechny možnosti, jak vyřešit nějaký problém.

Metoda pokusu a omylu se tento proces nazývá proto, že pokud již nemůžeme pokračovat dál (třeba v případě, že v bludišti dojdeme do slepé uličky), vrátíme se kus zpět a zkusíme jinou (zatím nevyzkoušenou) možnost. Takto postupně zkusíme každou možnost, a buď nalezneme námi hledané řešení, nebo se vrátíme až na výchozí pozici a zjistíme, že řešení neexistuje.

Backtracking bývá často realizován pomocí rekurze, ukážeme si to na příkladu hledání rozkladu zadané částky na mince o hodnotách 5 Kč a 3 Kč (všimněte si, že v takto omezeném peněžním systému nejde složit třeba částka 7 Kč). Naše funkce dostane jako parametr zbývající částku a zkusí rekurzivně provést rozklad na jednotlivé mince:

V jazyce C:

```
bool rozloz(int castka) {
        // Koncová podmínka rekurze
        if (castka == 0) return true;
        else if (castka < 0) return false;
        else if (rozloz(castka-5)) {
                printf(" 5 Kc");
                return true;
        } else if (rozloz(castka-3)) {
                printf(" 3 Kc");
                return true;
        } else return false;
}
```

V Pythonu:

```
def rozloz(castka):
    if castka == 0:
        return True
    elif castka < 0:
        return False
    elif rozloz(castka-5):
        print(" 5 Kc")
        return True
    elif rozloz(castka-3):
        print(" 3 Kc")
        return True
    else:
        return False
```

V každém kroku zkusíme nejdříve použít pětikorunovou minci a zavoláme se na zbylou částku, a když náš rozklad nevyjde, zkusíme v tomto kroku použít ještě tříkorunu. Takto se rozhodujeme v každém kroku rekurze a případně se vracíme z neúspěšných větví výpočtu a zkoušíme další možnosti.

Takovým postupem ale vyzkoušíme až exponenciálně mnoho možností ($\mathcal{O}(2^n)$), což není moc rychlé. Proto je doporučováno se backtrackování raději vyhnout, nebo ho nějak chytře vylepšit. Je však dobré o backtrackování vědět, protože existují problémy, které efektivněji řešit neumíme.

Rozděl a panuj

Jednou ze základních technik je rozdělení složitějšího problému na menší části, které opět můžeme rozdělit na menší a tak dále, dokud se nedostaneme k problémům tak malým, že je už umíme triviálně vyřešit.

Binární vyhledávání v poli

Představme si, že máme seřazené pole n prvků a chceme zjistit, jestli se v něm nachází prvek s hodnotou k. Určitě můžeme projít celé pole v lineárním čase (tím, že budeme brát jeden prvek za druhým a kontrolovat, zda je roven hodnotě k), ale to je zbytečně pomalé a nevyužívá toho, že máme pole seřazené.

Můžeme totiž začít s velkým problémem a ten postupně zmenšovat na stále menší a menší. Nejdříve hledáme k v celém poli. Podíváme se na jeho prostřední prvek:

- Pokud je roven k, jsme hotovi.
- Je-li větší než k, víme, že se k musí nacházet nalevo od něj. Můžeme tedy hledat znovu, ale tentokrát se omezit jen na levou polovinu pole.
- Analogicky, je-li menší než k, můžeme hledat jen v pravé polovině.

Když tímto postupným dělením problémů na menší dojdeme až k poli o velikosti jednoho prvku, stačí tento prvek jenom porovnat, dál už se pole nepokoušíme rozdělovat.

Jelikož se nám každým krokem problém zmenší na polovinu, maximálně po $\log n$ krocích se dostaneme na pole velikosti jedna. Říkáme, že algoritmus má *logaritmickou časovou složitost*, píšeme $\mathcal{O}(\log n)$.[13]

Prakticky postup provádíme tak, že si udržujeme levý a pravý okraj aktuálně zpracovávaného úseku a postupně je k sobě přibližujeme.

Ukázka hlavní smyčky v C:

```
int pole[] = {1,2,5,7,12,16,42};
int hledane = 8;
int L = 0, R = 6;
int x;
```

[13] Pokud není řečeno jinak, znamená pro nás v informatice značka log *dvojkový logaritmus*, což je funkce opačná k funkci 2^n a roste o hodně pomaleji než funkce lineární. Pro velká n platí: $1 < \log n < n$ a například $\log 2 = 1, \log 8 = 3, \log 1024 = 10$.

```
do {
    int prostredni = (L+R)/2;
    x = pole[prostredni];
    if (x == hledane)
        printf("Pole obsahuje hledane\n");
    else if (x < hledane)
        L = prostredni + 1;
    else
        R = prostredni;
} while (L < R && x != hledane);

if (x != hledane)
    printf("Hledane neni v poli\n");
```

Ukázka v Pythonu jako funkce vracející index prvku nebo -1, pokud hledané číslo nenalezne:

```
def bin_vyhled(pole, hledane,
               levy_index=0, pravy_index=None):
    if pravy_index is None:
        pravy_index = len(pole)
    while levy_index < pravy_index:
        prostredni = (levy_index +
                      pravy_index) // 2
        x = pole[prostredni]
        if x < hledane:
            levy_index = prostredni + 1
        elif x > hledane:
            pravy_index = prostredni
        else:
            return prostredni
    return -1

# Zavolání:
print(bin_vyhled([1,2,5,7, 8,12,16,42], 1))
```

Další aplikace

Další typickou aplikací postupu rozděl a panuj je například třídění posloupnosti pomocí *Mergesortu*. Ten v základu funguje tak, že každou posloupnost, kterou dostane k setřídění, rozdělí na poloviny a každou z nich setřídí rekurzivním zavoláním sebe sama. Zanořování se zastaví ve chvíli, kdy třídíme posloupnost délky jedna (ta už je z podstaty setříděná). Pak jen v každém kroku ze dvou setříděných menších posloupností vyrobí jejich sléváním setříděnou posloupnost dvojnásobné délky.

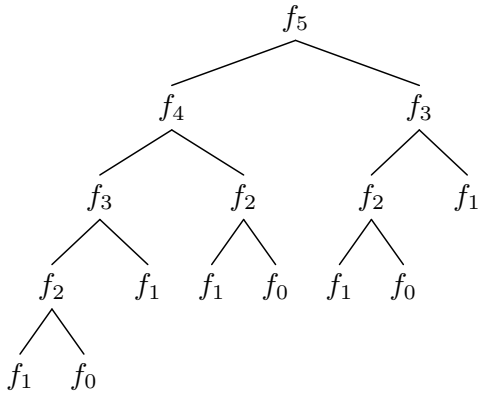
Více se o metodě Rozděl a panuj můžete dozvědět ve stejnojmenné kuchařce.[14]

Předpočítané mezivýsledky

Motivací k této kapitole je následující motto: „Proč počítat něco vícekrát, když nám to stačí spočítat jednou a zapamatovat si to?“.

Velmi často se totiž setkáváme s tím, že něco počítáme stále dokola. Jako příklad si můžeme připomenout naši rekurzivní implementaci počítání Fibonacciho čísel zmíněnou výše.

Když se podíváme na výpočet čísla `fib(5)`, vidíme, že pro něj voláme `fib(4)` a `fib(3)`, `fib(4)` volá `fib(3)` a `fib(2)`, `fib(3)` volá `fib(2)` a `fib(1)` a tak dále. Všimli jste si, kolikrát se nám tyhle výpočty opakují? Některá Fibonacciho čísla spočteme totiž zbytečně mnohokrát.



Kdybychom si je namísto opakovaného počítání někde pamatovali, mohli bychom pak odpověď na dotaz na již vypočtené číslo vytáhnout jako králíka z klobouku v konstantním čase. Zavedením jednoho globálního pole, ve kterém si tyto hodnoty pro jednotlivá n budeme pamatovat, nám sníží časovou složitost z $\mathcal{O}(2^n)$ na pěkných $\mathcal{O}(n)$. Takovému postupu se obecně říká *dynamické programování*.

Dynamické programování

Nejprve uveďme na pravou váhu výraz „dynamické“ v názvu. Nevystihuje tak úplně podstatu této techniky a jeho historické pozadí je celkem složité, avšak dnes je tento název již tak zažitý, že se už pravděpodobně nezmění.

Slovo „dynamické“ částečně odkazuje na to, že se dynamicky (za běhu programu) postupně staví řešení jednodušších problémů, která jsou následně použita pro řešení složitějších. Jeho hlavní podstatou je tedy ukládání a opětovné použití již jednou vypočtených údajů.

Hodí se na úlohy, které se dají dělit na podúlohy, které jsou si podobné a mohou se opakovat. Výsledky takovýchto podúloh si poté ukládáme a při dotazu na stejnou podúlohu vrátíme jen uložený výsledek a výpočet již neprovádíme.

Pro další prohloubení znalostí můžete na našem webu nahlédnout do další kuchařky, tentokrát nesoucí (překvapivě) název Dynamické programování.[15]

Prefixové součty

Velmi často se nám hodí si ještě před samotným výpočtem předpočítat a uložit nějaké hodnoty, které poté použijeme.

Představme si například problém nalezení souvislého úseku s největším součtem v nějaké posloupnosti kladných i záporných čísel. Že to není úplně jednoduchý příklad, si ukažme na následující posloupnosti:

$$1, -2, 4, 5, -1, -5, 2, 7$$

Máme zde dvě ryze kladné souvislé posloupnosti, každou se součtem 9 ($4, 5$ a $2, 7$). Ale přesto je výhodnější vzít i nějaké záporné hodnoty a vytvořit tak souvislou posloupnost o součtu 12 (zkuste ji nalézt).

Mohlo by nás napadnout, že prostě zkusíme vzít všechny možné začátky a všechny možné konce. To nám dává $\mathcal{O}(n^2)$ možných posloupností (máme n možných začátků a ke každému z nich řádově n možných konců), pro každou posloupnost si spočteme součet (to zvládneme v $\mathcal{O}(n)$) a budeme si pamatovat ten největší nalezený. Celý náš postup tak trvá $\mathcal{O}(n^3)$.

To není pro takhle jednoduchou úlohu zrovna ten nejpěknější čas, zkusme ho zlepšit. Ukážeme si, jak vypočítat sou-

[14] `http://ksp.mff.cuni.cz/viz/kucharky/rozdel-a-panuj`
[15] `http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani`

čet libovolné posloupnosti v konstantním čase. Celý princip je vlastně až kouzelně jednoduchý, ale zároveň velmi mocný. Na začátku výpočtu si do pomocného pole P stejné délky jako posloupnost na vstupu (té říkejme S) uložíme takzvané *prefixové součty:* i-tý prefixový součet je součet prvních $i+1$ prvků S, neboli $P[i] = S[0] + S[1] + \ldots + S[i]$.

Pro náš ukázkový případ a pro vstupní pole označené S by to dopadlo takto:

i	-1	0	1	2	3	4	5	6	7
$S[i]$		1	-2	4	5	-1	-5	2	7
$P[i]$	0	1	-1	3	8	7	2	4	11

Pole prefixových součtů umíme získat v linerárním čase – prostě jen od začátku procházíme vstupní pole, počítáme si průběžný součet a ten zapisujeme.

Součet libovolného úseku $a \ldots b$ pak získáme v konstantním čase jako prefixový součet od začátku do indexu b minus prefixový součet od začátku do indexu a. Zapsáno programově to pak je:

```
soucet = P[b] - P[a-1];
```

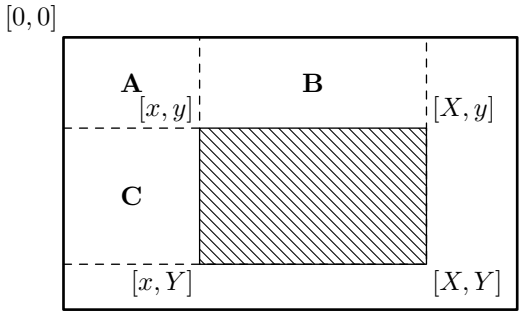
To nám umožňuje snížit čas potřebný na řešení této úlohy na $\mathcal{O}(n^2)$. To je už lepší čas; prozradíme však, že tuto úlohu lze řešit dokonce v lineárním čase, ale to je již nad rámec této kuchařky.

Dvourozměrné prefixové součty

Prefixové součty se dají zobecnit i do více rozměrů, ale princip je vždy stejný. Například dvourozměrné prefixové součty u matice fungují tak, že si předpočítáme součty podmatic začínajících levým vrchním políčkem a končící na indexu $[x, y]$.

Z toho je vidět, že prefixový součet zpravidla obsadí stejně velký prostor jako původní data, v tomto případě tedy budeme mít matici hodnot prefixových součtů končících na zadaných souřadnicích. Jak ale získat součet nějaké podmatice, která se nachází někde „uprostřed“ naší matice?

Použijeme stejný princip jako u jednorozměrného případu: Přičteme větší část, kterou chceme započítat, a odečteme od ní části, které započítat nechceme. Pro případ podmatice začínající vlevo nahoře na pozici $[x, y]$ a končící napravo dole na $[X, Y]$ to ilustruje následující obrázek:



Nejdříve přičteme celý prefixový součet končící na pozici $[X, Y]$. Tím jsme ale započítali i části A, B a C z obrázku, které započítat nechceme. Tak odečteme prefixové součty končící na indexech $[X, y]$ a $[x, Y]$. Ale pozor, teď jsme odečetli jednou $A + B$ a jednou $A + C$, tedy část A (prefixový součet končící na pozici $[x, y]$) jsme odečetli dvakrát, musíme ji proto ještě jednou přičíst.

Celý vzorec tedy vypadá takto:

```
soucet = P[X,Y] - P[X,y] - P[x,Y] + P[x,y];
```

Tento princip přičítání a odečítání se dá zobecnit i pro libovolné vyšší rozměry, ale chce to již trošku představivosti, co se má přičíst a kolikrát. Říká se tomu také *princip inkluze a exkluze* a najde použití nejen u vícerozměrných prefixových součtů.

Vyvážení délky předvýpočtu a hlavního výpočtu

Správně vyvážit, kolik času můžeme věnovat na předvýpočet a kolik času na hlavní výpočet, je velice důležitá věc a spousta i zkušenějších řešitelů v tom občas chybuje. Přitom to při troše počítání není vůbec nic složitého.

Jako první je potřeba vědět, kolikrát nám předvýpočet během běhu programu pomůže. Předvýpočtem si totiž vybudujeme za nějaký čas určitou datovou strukturu, pomocí které pak dokážeme rychle odpovídat na zadané dotazy.

Označme si počet takovýchto dotazů, které program za běhu dostane, jako Q. Buď to může být hodnota přímo ze zadání typu „Zkonstruujte datovou strukturu pro n hodnot, která zvládne rychle odpovídat na dotazy daného typu, a očekávejte řádově m dotazů“, nebo se může jednat o nějaký interní dotaz v rámci běhu programu (příklad interního dotazu je třeba výše uvedený algoritmus na hledání souvislé podposloupnosti s co největším součtem, který se za běhu ptal na součty nějakých úseků).

Dále si označme jako $\mathcal{O}_p$ čas, který nám zabere předvýpočet a jako $\mathcal{O}_q$ čas, který nám ušetří každý předvypočítaný dotaz. Celkový čas, který ušetříme, je pak vlastně $Q \cdot \mathcal{O}_q$. Pokud je tento čas řádově větší než $\mathcal{O}_p$, předvýpočet má smysl.

Naopak nemá smysl trávit předvýpočtem řádově více času, než by trval samotný výpočet bez použití předpočítaných hodnot.

Jako příklad uvažujme problém o velikosti n, u kterého máme tři možnosti, které můžeme zvolit. Můžeme buď předvýpočet úplně vynechat a na každý dotaz odpovídat v čase $\mathcal{O}(n)$, nebo provést předvýpočet v čase $\mathcal{O}(n \log n)$ a poté odpovídat na každý dotaz v čase $\mathcal{O}(\log n)$, nebo provést předvýpočet v čase $\mathcal{O}(n^2)$ a pak odpovídat v čase $\mathcal{O}(1)$ na dotaz.

Kdy se nám co hodí?

- Pokud bychom dostali jen jeden dotaz, nemá smysl si cokoliv předpočítávat a odpovíme jednou v čase $\mathcal{O}(n)$.
- Pokud bude dotazů řádově n, má smysl použít první předvýpočet. Pak budeme mít čas na předvýpočet i na samotný výpočet $\mathcal{O}(n \log n)$, což je optimum.
- Naopak pokud by dotazů bylo řádově n^2 nebo víc, tak se nám již první předvýpočet nevyplatí, dostali bychom se totiž na čas $\mathcal{O}(n^2 \log n)$. Zde se hodí použít druhý, delší předvýpočet a pak se dostat na časovou složitost $\mathcal{O}(n^2 + n^2 \cdot 1) = \mathcal{O}(n^2)$.

Hladové algoritmy

Věřte nebo ne, ale i počítač se někdy cítí hladový. Po namáhavé práci mu můžeme dopřát to potěšení, aby si ukousl co největší kus dat. A ukážeme, že někdy je to i ku prospěchu. Řeč bude o *hladových algoritmech*.

Takovými algoritmy rozumíme ty, které hledají řešení celé úlohy po jednotlivých krocích a splňují následující dvě podmínky:

- V každém kroku zvolí lokálně nejlepší řešení.
- Provedené rozhodnutí již nikdy neodvolává (tedy nebacktrackuje).

Lokálně nejlepší řešení je takové, které v aktuálním kroku vybere tu možnost, která nám na tomto místě nejvíce pomůže (bez jakéhokoliv ohledu na globální stav). Může to být třeba nejvyšší hodnota, nebo nejkratší cesta v grafu.

Pokud ale od hladového algoritmu chceme, aby nám našel globálně nejlepší řešení, musí naše úloha splnit předpoklad, že si výběrem lokálně nejlepšího řešení nezhoršíme to globální. Tento předpoklad se nedá formulovat obecně a je nutné se nad ním zamyslet zvlášť u každé úlohy.

Příklady hladových algoritmů

První hladovou úlohou bude (jak jinak) automat na jídlo vracející mince. Automat by měl vracet peníze nazpět tak, aby vrátil daný obnos v co možná nejmenším počtu mincí. Pro náš měnový systém (máme mince hodnot 1, 2, 5, 10, 20 a 50 Kč) lze tuto úlohu řešit hladovým algoritmem – v každém kroku algoritmu vrátíme tu největší minci, kterou můžeme (tedy pro vrácení 42 Kč to bude $42 = 20 + 20 + 2$ Kč).

Měnové systémy většiny států jsou postavené tak, aby fungovaly takto pěkně, neplatí to ale obecně. Zkusme si vrátit 42 Kč, pokud bychom měli jen mince hodnoty 20, 10 a 4 Kč. Správným řešením je $42 = 20 + 10 + 4 + 4 + 4$ Kč, hladový algoritmus by ale zkusil vrátit $42 = 20 + 20 + \ldots$ a tady by selhal.

Dále se velmi často dají hladovým algoritmem řešit nějaké úlohy přidávání nebo odebírání skupin prvků. Typickým příkladem je třeba rozvržení naplánovaných přednášek do učeben. Seřadíme si začátky přednášek podle času a postupně bereme jednu za druhou a umísťujeme je do volných učeben s nejnižším číslem.

Tím jsme si určitě nic nerozbili, protože v nějaké učebně přednáška být musí. Určitě budeme potřebovat tolik učeben, kolik je maximálně přednášek v jeden čas, a díky tomu si umístěním přednášky do nějaké učebny nezablokujeme místo pro jinou přednášku, jelikož nám vždy zbude dostatek volných učeben.

Kdybychom ale naopak měli pevně zadaný počet učeben a chtěli jsme do nich umístit co možná nejvíce přednášek, nejedná se již o úlohu řešitelnou hladovým algoritmem, v takovém případě je potřeba zvolit nějaký chytřejší postup.

Závěr

Doufám, že jste si z tohoto rozsáhlého textu odnesli nějaké nové znalosti a poznatky, které vám pomohou nejen v řešení KSP.

Pokud jste začínajícími řešiteli, zkuste s pomocí kuchařky vyřešit několik lehčích úloh a jejich řešení poslat – nově nabyté znalosti je totiž nejlepší co nejdříve protrénovat. Nic si nedělejte z toho, pokud napoprvé nevyřešíte všechno, s postupným zkoušením se budou vaše znalosti jen zlepšovat. Zkušenější řešitelé možná v kuchařce nalezli nějaké ujasnění pojmů, či si některé techniky osvěžili.

A pokud tento text považujete za dobrý, budeme jen rádi, pokud ho doporučíte svým kamarádům a spolužákům, kteří chtějí s programováním začít.

Úvodním kurzem vaření podle kuchařky vás provedl

Jirka Setnička